

Refs AK+AB CA PFO20079



CITED BY APPLICANT

①9 BUNDESREPUBLIK
DEUTSCHLAND



DEUTSCHES
PATENT- UND
MARKENAMT

⑫ Offenlegungsschrift

⑩ DE 199 45 992 A 1

⑤ Int. Cl. 7:
G 06 F 9/455
G 06 F 9/45

⑳ Aktenzeichen: 199 45 992.4
㉔ Anmeldetag: 24. 9. 1999
㉔ Offenlegungstag: 4. 5. 2000

US Pat.
5002147969
corres.

DE 199 45 992 A 1

③③ Unionspriorität:
176112 21. 10. 1998 US
㉒ Anmelder:
Fujitsu Ltd., Kawasaki, Kanagawa, JP
㉒ Vertreter:
W. Seeger und Kollegen, 81369 München

㉒ Erfinder:
Lethin, Richard A., New York, US; Bank, Joseph A.,
New York, US; Garrett, Charles D., New York, US;
Wada, Mikayo, Kawasaki, Kanagawa, JP; Sakurai,
Mitsuo, Kawasaki, Kanagawa, JP

Die folgenden Angaben sind den vom Anmelder eingereichten Unterlagen entnommen

Prüfungsantrag gem. § 44 PatG ist gestellt

⑤④ Dynamisch optimierender Objektcode-Übersetzer zur Architecturemulatun und dynamisches optimierendes Objektcode-Übersetzungsverfahren

⑤⑦ Optimierendes Objektcode-Übersetzungssystem sowie Verfahren zur Durchführung einer dynamischen Kompilierung und Übersetzung eines Ziel-Objektcodes auf einem Quellen-Betriebssystem, während die Optimierung vorgenommen wird. Die Kompilierung und Optimierung des Zielcodes werden dynamisch in Echtzeit ausgeführt. Ein Kompilierer führt eine Analyse und Optimierungen durch, welche die Emulation gegenüber einer auf einer Schablone basierenden Übersetzung und Interpretation verbessern, so daß ein Hostprozessor, der Instruktionen höherer Ordnung, wie 32 Bit-Instruktionen, verarbeitet, einen Zielprozessor emulieren kann, der Instruktionen niedrigerer Ordnung, wie 16 Bit- und 8 Bit-Instruktionen, verarbeitet. Der optimierende Objektcode-Übersetzer erfordert keine Kenntnis eines statischen Programmflußgraphen oder von Speicherzellen vor der Laufzeit. Ferner erfordert der optimierende Objektcode-Übersetzer keine Kenntnis des Orts aller Verbindungspunkte in den Ziel-Objektcode vor der Ausführung. Während der Programmausführung zeichnet ein Übersetzer Verzweigungsoperationen auf. Die Protokollierung von Informationen identifiziert Instruktionen und Instruktionsverbindungspunkte. Wenn eine Anzahl von Malen, die eine Verzweigungsoperation ausgeführt wird, eine Schwelle überschreitet, wird das Ziel der Verzweigung ein Startparameter für die Kompilierung, und Codeteile zwischen Startparametern werden als Segmente definiert. Ein Segment kann unvollständig sein, wodurch eine Modifikation oder ...

DE 199 45 992 A 1

HINTERGRUND DER ERFINDUNG

Die vorliegende Erfindung bezieht sich auf die Technik von Objektcode-Übersetzern, die auf einem Hostverarbeitungssystem operieren, um ein zweites Betriebssystem zu emulieren. Insbesondere betrifft die vorliegende Erfindung die Technik dynamischer Objektcode-Übersetzer, die eine Analyse und Berechnung eines Original-Objektcode-Instruktionsatzes in Echtzeit während der Ausführung auf einem Hostprozessor, der einen Hostprozessor-Objektcode-Instruktionsatz aufweist, durchführen.

Auf dem Gebiet von Objektcode-Übersetzern ist es notwendig, einen Objektcode, der für einen Computer entwickelt wurde, auf einem anderen Computer mit einer unterschiedlichen Computerarchitektur zu konvertieren. Konvertierungsverfahren für einen derartigen Objektcode enthalten ein herkömmliches Verfahren, das als "statisches Objektcode-Konvertierungsverfahren" bezeichnet wird, und bei dem Instruktionsanweisungen vor der Ausführung zuerst in einen Objektcode einer zweiten Architektur konvertiert werden. Ein zweites Verfahren ist ein "dynamisches Objektcode-Konvertierungsverfahren", bei dem ein erster Objektcode in einen zweiten Objektcode während der Ausführung von Instruktionen konvertiert wird.

In der Technik statischer Objektcode-Konvertierungsverfahren wird die Ausführungszeit von der für die Konvertierung erforderlichen Zeit nicht beeinflusst. Die physische Größe des konvertierten Objektcodes wird jedoch bei der Ausführung der statischen Objektcode-Konvertierung groß. Mit anderen Worten nimmt beim statischen Objektcode-Konvertierungsverfahren eine Anzahl von Betriebssystemschritten im konvertierten Objektcode unweigerlich zu. Folglich entsteht insofern ein Problem, als sich die Leistung des konvertierten Objektcodes verschlechtert, und es zu Ineffizienzen kommt.

Andererseits wird beim dynamischen Objektcode-Konvertierungsverfahren die Größe des konvertierten Objektcodes relativ klein im Vergleich zum statischen konvertierten Objektcode. Das herkömmliche dynamische Objektcode-Konvertierungsverfahren weist jedoch insofern ein Problem auf, als alle Objekte, einschließlich selten verwendeter Objekte, konvertiert werden. Mit anderen Worten kann das herkömmliche dynamische Objektcode-Konvertierungsverfahren Objekte, die mehrere Male ausgeführt werden, nicht effizient erkennen, und dadurch nimmt die Zeit, die für die Konvertierung des Original-Objektcodes erforderlich ist, auf Kosten der Effizienz zu.

KURZFASSUNG DER ERFINDUNG

Demgemäß ist es ein Objekt der vorliegenden Erfindung, einen Objektcode-Übersetzer vorzusehen, der die bekannten Probleme eliminiert, während er eine dynamische Optimierung des übersetzten Objektcodes vorsieht.

Es ist ein weiteres Objekt der vorliegenden Erfindung, ein Hauptprogramm zu profilieren, bis ein Kompilierer das Kompilieren vollendet hat, wobei dieses Profil vom Kompilierer zum Kompilieren und Optimieren des Programms verwendet wird.

Es ist noch ein weiteres Objekt der vorliegenden Erfindung, während der dynamischen Optimierung und Kompilierung vom nicht-übersetzten Code zum übersetzten zu springen.

Es ist noch ein weiteres Objekt der vorliegenden Erfindung, einen dynamischen optimierenden Objektcode-Übersetzer mit einer Software-Rückkopplung vorzusehen, der die Differenz zwischen einer Anzahl von Übersetzungsanforderungen, die an den Kompilierer gesendet werden, und einer Anzahl vollendeter Übersetzungen berechnet.

Es ist noch ein weiteres Objekt der vorliegenden Erfindung, eine dynamische Übersetzung eines Computerprogramms in einer Maschinensprache in eine andere Maschinensprache vorzusehen, während das Programm läuft.

Ferner ist ein Objekt der vorliegenden Erfindung, einen dynamischen Objektcode-Übersetzer vorzusehen, der Segmente zur Übersetzung aus einer Vielzahl von Startparametern bestimmt, die Verzweigungen in einem Quellen-Objektcode entsprechen.

Die Objekte der vorliegenden Erfindung werden durch ein Computerarchitektur-Emulationssystem erreicht, welches eine Quellen-Computerarchitektur auf einer Ziel-Computerarchitektur emuliert, mit einem Interpretierer zum individuellen Übersetzen eines Quellen-Objektcodes in einen entsprechenden übersetzten Objektcode und zum Bestimmen einer Anzahl von Ausführungen von Verzweigungsstrukturen im Quellen-Objektcode; und einem Kompilierer zum Gruppieren von Instruktionen des Quellen-Objektcodes in ein Segment, wenn eine Anzahl von Ausführungen einer entsprechenden Verzweigungsstrukturen eine Schwellenanzahl überschreitet, und zum dynamischen Kompilieren des Segments.

Ferner werden die Objekte der vorliegenden Erfindung durch ein Computerarchitektur-Emulationssystem erreicht, welches eine Quellen-Computerarchitektur auf einem Ziel-Computerarchitektursystem emuliert, mit einer Vielzahl von Interpretierern zum individuellen Übersetzen eines Quellen-Objektcodes in einen entsprechenden übersetzten Objektcode, wobei jeder der Vielzahl von Interpretierern Quellen-Objektcode-Verzweigungsinformationen in Echtzeit profiliert, während er übersetzte Objektcode-Instruktionen ausführt; und einem Kompilierer zum Gruppieren von Quellen-Objektcode-Instruktionen von irgendeinem der Vielzahl von Interpretierern in Segmente auf der Basis entsprechender Verzweigungsinstruktionen im Quellen-Objektcode und zum dynamischen Kompilieren der Segmente des Quellen-Objektcodes, wenn die entsprechende Verzweigungsstrukturen größer ist als eine Schwellenanzahl.

Noch weitere Objekte der vorliegenden Erfindung werden durch ein Computerarchitektur-Emulationssystem erreicht, das eine Quellen-Computerarchitektur auf einem Ziel-Computerarchitektursystem emuliert, mit einem Interpretierer zum individuellen Übersetzen eines Quellen-Objektcodes in einen entsprechenden übersetzten Objektcode, wobei der Interpretierer Verzweigungsinstruktionen des Quellen-Objektcodes durch das Speichern einer Anzahl von Ausführungen für jede Verzweigungsstrukturen und Vergleichen der Anzahl von Ausführungen mit einer Schwellenanzahl profiliert, so daß Verzweigungsstrukturen, welche die Schwellenanzahl überschreiten, Startparameter sind; und einem Kompilierer zum Gruppieren der Quellen-Objektcode-Instruktionen in Segmente auf der Basis der Startparameter und dynamischen Kompilieren der Segmente des Quellen-Objektcodes während der Übersetzung und Profilierung durch den Interpretierer.

Zusätzliche Objekte der vorliegenden Erfindung werden durch ein Mehraufgaben-Computerarchitektur-Emulationssystem erreicht, welches eine Quellen-Computerarchitektur auf einer Mehraufgaben-Ziel-Computerarchitektur emuliert, mit einer Interpretieraufgabe zum individuellen Übersetzen eines Quellen-Objektcodes in einen entsprechenden übersetzten Objektcode und zum Bestimmen einer Anzahl von Ausführungen von Verzweigungsanweisungen im Quellen-Objektcode; und einer Kompilieraufgabe, die mit dem Interpretierer auf der Mehraufgaben-Ziel-Computerarchitektur operiert, zum Gruppieren von Anweisungen des Quellen-Objektcodes in ein Segment, wenn eine Anzahl von Ausführungen einer entsprechenden Verzweigungsanweisung eine Schwellenanzahl überschreitet, und zum dynamischen Kompilieren des Segments.

KURZE BESCHREIBUNG DER ZEICHNUNGEN

Diese und andere Objekte und Vorteile der vorliegenden Erfindung gehen aus der folgenden Beschreibung der bevorzugten Ausführungsformen in Verbindung mit den beigezeichneten Zeichnungen hervor und werden dadurch besser verständlich, wobei:

Fig. 1 ein Blockbild einer höheren Architektur eines OOC-Systems gemäß einer bevorzugten Ausführungsform der vorliegenden Erfindung ist;

Fig. 2 ein Flußdiagramm ist, das Hauptkomponenten einer optimierenden Objektcode-Übersetzung zusammen mit einem Steuerfluß zur Kompilierung eines Abschnitts eines Originalcodes veranschaulicht;

Fig. 3 ein Flußdiagramm ist, das den Steuerfluß in einer optimierenden Objektcode-Übersetzung während der normalen Ausführung veranschaulicht;

Fig. 4 eine schematische Darstellung ist, die einen OOC-Puffer für eine Einstellung von Variablen veranschaulicht;

Fig. 5a, 5b und 5c schematische Darstellungen sind, welche die Struktur einer Übersetzungstabelle veranschaulichen;

Fig. 6 ein Blockbild eines Interpretierers zum Einspringen in ein Segment und Verlassen desselben ist;

Fig. 7 ein Blockbild eines Kompilierungsverfahrens ist, um ein Segment zu schaffen, um das Segment durch einen Interpretierer erreichbar zu machen, um alte Segmente unerreichbar zu machen, und um alte Segmente zu löschen;

Fig. 8 ein Blockbild ist, das eine Struktur eines Verzweigungsdatensatzes bzw. engl. `BRANCH_RECORD` veranschaulicht;

Fig. 9 eine schematische Darstellung ist, welche eine Struktur eines Verzweigungsprotokolls als Teil einer großen Hash-Tabelle veranschaulicht, die `BRANCH_RECORDS` speichert;

Fig. 10 eine schematische Darstellung ist, welche eine Struktur eines L1-Cache veranschaulicht, der ein zweidimensionales Array von `BRANCH_L1_RECORDS` darstellt;

Fig. 11 eine schematische Darstellung ist, die ein Verfahren zur Ausführung des Betriebs des L1-Cache durch einen Interpretierer ist;

Fig. 12 eine schematische Darstellung ist, die eine allgemeine Struktur eines Kompilierers gemäß einer Ausführungsform der vorliegenden Erfindung veranschaulicht;

Fig. 13 eine schematische Darstellung ist, die ein Beispiel eines Blockwählers gemäß einer Ausführungsform der vorliegenden Erfindung veranschaulicht;

Fig. 14 ein Blockbild einer Codeübersicht mit zwei externen Einsprungpunkten ist, wobei Füllzeichen zwischen der Einsprung- bzw. engl. `ENTRY`-Anweisung und der Sprung-nach- bzw. engl. `GOTO`-Anweisung eingefügt wurden;

Fig. 15 ein Blockbild ist, das ein `OASSIGN`-Einfügungsbeispiel veranschaulicht;

Fig. 16 ein Blockbild ist, das ein Beispiel einer Totcodeeliminierung und Adressenprüfungseliminierung veranschaulicht;

Fig. 17 ein Blockbild eines Beispiels einer Adressenprüfungseliminierung ist;

Fig. 18 ein Blockbild eines Beispiels einer gemeinen Teilausdruckeliminierung bzw. engl. `Common Subexpression Elimination` ("CSE") ist;

Fig. 19 ein Blockbild einer Kopierpropagation bzw. engl. `Copy Propagation` ist;

Fig. 20 insbesondere ein Beispiel einer konstanten Faltung bzw. engl. `Constant Folding` veranschaulicht;

Fig. 21 insbesondere ein Beispiel des obigen Prozesses gemäß einer Ausführungsform der vorliegenden Erfindung veranschaulicht, der eine Vergleichsinfrastruktur aufweist;

Fig. 22 insbesondere ein Beispiel der Codegenerierung für dieselbe Anweisung mit unterschiedlichen umgebenden Anweisungen veranschaulicht;

Fig. 23 eine Systemkonfiguration gemäß der zweiten Ausführungsform der vorliegenden Erfindung veranschaulicht, die zur dynamischen optimierenden Objektcode-Übersetzung verwendet wird;

Fig. 24 eine Systemkonfiguration gemäß der dritten Ausführungsform der vorliegenden Erfindung veranschaulicht, die zur gleichzeitigen dynamischen Übersetzung verwendet wird;

Fig. 25 eine Differenz zwischen der Kombination eines Interpretierers und Kompilierers, beispielsweise während der Ausführung als eine Aufgabe, und ihrer Trennung, beispielsweise in verschiedene Aufgaben, gemäß einer dritten Ausführungsform der vorliegenden Erfindung veranschaulicht;

Fig. 26 eine Übersetzungstabelle gemäß einer vierten Ausführungsform der vorliegenden Erfindung veranschaulicht, die zur Aufzeichnung verwendet wird, welche Anweisungen übersetzbar sind und welche nicht;

Fig. 27 veranschaulicht, wie das Verfahren die Belastung der Profilierung auf den Emulator gemäß einer vierten Ausführungsform der vorliegenden Erfindung reduziert;

Fig. 28 eine allgemeine Strukturdarstellung eines dynamischen Übersetzungssystems mit getrenntem Interpretierer und Kompilierer gemäß einer fünften Ausführungsform der vorliegenden Erfindung veranschaulicht;

Fig. 29 Komponenten eines Software-Rückkopplungsmechanismus gemäß einer fünften Ausführungsform der vorliegenden Erfindung veranschaulicht;

Fig. 30 veranschaulicht, wie gemäß einer sechsten Ausführungsform der vorliegenden Erfindung eine Warteschlange zum Halten von Übersetzungsanforderungen verwendet wird, während die Übersetzungsaufgabe belegt ist;

Fig. 31 veranschaulicht, wie die OOC-Anforderungswarteschlange kostengünstig mit gemeinsam genutzte Speicheranforderungen mit Systemaufrufanforderungen gemäß einer sechsten Ausführungsform der vorliegenden Erfindung kombiniert;

Fig. 32 zeigt, wie gemäß einer siebenten Ausführungsform der vorliegenden Erfindung ein dynamischer Übersetzer Seitenfehler verursachen kann, die während der normalen Ausführung der Quelleninstruktionen nicht auftreten würden;

Fig. 33 den Algorithmus zur Behebung von Seitenfehlern während der Übersetzung und die Fortsetzung der Übersetzung gemäß einer siebenten Ausführungsform der vorliegenden Erfindung zeigt;

Fig. 34 ein Muster eines Steuerflusses in einem dynamischen Übersetzungssystem mit einem Verzweigungsprofilierer gemäß einer achten Ausführungsform der vorliegenden Erfindung veranschaulicht;

Fig. 35 veranschaulicht, wie gemäß einer neunten Ausführungsform der vorliegenden Erfindung der dynamische Übersetzer Verzweigungsprofilinformationen verwendet, um die Ausführungswahrscheinlichkeit eines Basisblocks zu berechnen.

DETAILLIERTE BESCHREIBUNG DER BEVORZUGTEN AUSFÜHRUNGSFORMEN

Nun wird detailliert auf die bevorzugten Ausführungsformen der vorliegenden Erfindung bezuggenommen, wovon Beispiele in den beigeschlossenen Zeichnungen veranschaulicht sind, wobei sich ähnliche Bezugszahlen durchgehend auf ähnliche Elemente beziehen.

ERSTE AUSFÜHRUNGSFORM DER VORLIEGENDEN ERFINDUNG

I. SYSTEMÜBERBLICK

Die vorliegende Erfindung bezieht sich allgemein auf einen optimierenden Objektcode-Übersetzer, nachstehend "OOCT", der eine dynamische Kompilierung eines Mikroprozessor-Instruktionssatzes als Teil eines Computerarchitektur-Emulationssystems durchführt. Die Kompilierung ist dynamisch, da es keinen einfachen Zugriff auf den Applikationsinstruktionssatz vor der Laufzeit gibt. Die Verwendung eines Kompilierers als Teil des Objektcode-Übersetzungssystems ermöglicht es dem System, eine Analyse und Optimierungen durchzuführen, welche die Leistung der Emulation gegenüber auf Schablonen basierenden Übersetzungen und auf Schablonen basierenden Interpretationen zu verbessern.

Der Hostprozessor für die Emulation ist vorzugsweise ein im Handel erhältlicher Prozessor, wie der Intel Pentium Pro. Die Architektur des Pentium Pro-Instruktionssatzes erleichtert die Manipulation verschiedener Datengrößen, und erleichtert dadurch die Emulation sowohl von 16 Bit- als auch 8 Bit-Objektcode-Instruktionen. Die 16 Bit- und 8 Bit-Objektcode-Instruktionen können für eine Originalapplikation auf einem zweiten Prozessor, wie einem Prozessor der K-Serie von Fujitsu, ausgebildet sein.

Die Durchführung sinnvoller Optimierungen vom Kompilierer-Typ ist nur durch die Kenntnis eines Instruktionsflußgraphen möglich. In einem traditionellen Kompilierer ist der Flußgraph gegeben und gut definiert, da die gesamte Routine einer vollständigen Sprachanalyse unterzogen wird, bevor die Optimierung beginnt. Bei einem OOC ist dies nicht der Fall. Bevor das Programm läuft, ist der Ort der Instruktionen im Speicherbild unbekannt. Dies ist darauf zurückzuführen, daß die Instruktionen eine variable Länge mit willkürlichen intervenierenden Sätzen von Nicht-Instruktionsdaten aufweisen. Der Ort der Instruktionen ist unbekannt, ebenso wie der Ort aller Verbindungspunkte in die Instruktionen.

Daher muß zur Bestimmung des Flußgraphen das Programm laufen gelassen werden. Zuerst läßt ein Interpretierer das Programm laufen. Während der Interpretierer das Programm ausführt, informiert der Interpretierer den OOC jedesmal, wenn er eine Verzweigungsoperation durchführt. Diese Protokollierung von Informationen identifiziert einige der Instruktionen und einige der Verbindungspunkte. Während das Programm läuft, werden die Informationen über den Flußgraphen zwar vollständiger, jedoch niemals ganz vollständig. Das OOC-System ist ausgebildet, mit Teilinformationen über den Flußgraphen zu arbeiten: die Optimierung wird an potentiell unvollständigen Flußgraphen durchgeführt, und das System ist ausgebildet, das Ersetzen des optimierten Codes mit zunehmender Verfügbarkeit von Informationen zu gestatten.

Die dynamische Kompilierung wählt auf der Basis der vom Interpretierer gesammelten Profilinformatoren, welche Teile des Textes zu optimieren sind. Wenn die Anzahl von Malen, die irgendeine Verzweigung ausgeführt wird, eine Schwellenanzahl überschreitet, wird das Ziel dieser Verzweigung ein Startparameter für die Kompilierung. Der Startparameter ist ein Startpunkt für eine Sprachanalyse eines Teils der K-Instruktionen, die als Einheit zu kompilieren sind. Dies wird Segment genannt.

Ein Segment enthält Hostprozessorinstruktionen, die aus der Optimierung der Originalprozessorinstruktionen vom Startparameter resultieren. Ein Segment wird als Einheit installiert und deinstalliert. Wenn der Interpretierer den OOC aufruft, um über eine Verzweigung zu informieren, kann der OOC wählen, die Steuerung in das Segment zu transferieren, wenn der Code für das Ziel existiert. Ähnlich kann das Segment einen Code zum Transferieren der Steuerung zurück zum Interpretierer enthalten.

Ein Segment selbst kann unvollständig sein, so daß das Segment nur einen Teilsatz der möglichen Flußpfade vom Originalprogramm repräsentiert. Diese unvollständige Repräsentation beeinflusst jedoch nicht den korrekten Betrieb der Emulation. Wenn ein neuer, unvorhergesehener Flußpfad durch den Originalcode auftritt, springt der Steuerfluß zurück zum Interpretierer. Später kann dasselbe Segment ersetzt werden, um den neuen Steuerfluß zu berücksichtigen.

II. OOC-CODESTRUKTUR

Gemäß einer Ausführungsform der vorliegenden Erfindung kann der OOC unter einer herkömmlichen Betriebssystemumgebung wie Windows laufen. Gemäß einer zweiten Ausführungsform der vorliegenden Erfindung kann der OOC jedoch eingerichtet sein, mit einer Emulations-Firmware auf einem zweiten Betriebssystem verbunden zu wer-

den, wie dem KOI-Betriebssystem von Fujitsu.

III. ARCHITEKTUR

Fig. 1 veranschaulicht die höhere Architektur des OOC-Systems 100. Fig. 1 veranschaulicht zwei Aufgaben, nämlich einen Interpretierer 110 und einen Kompilierer 104. Der Interpretierer 110 und der Kompilierer 104 operieren gleichzeitig unter einem Mehraufgaben-Betriebssystem. Die zwei Aufgaben können beide durch einen Verzweigungsprotokollierer 112 auf ein Verzweigungsprotokoll zugreifen, und können auch auf die kompilierten Codesegmente 108 zugreifen. Außerdem kann der Interpretierer 110 Kompilierungsanforderungen an den Kompilierer 104 senden. Eine vollständigere Beschreibung der Kommunikation zwischen den beiden Aufgaben erfolgt im nachstehend angegebenen Kommunikationsabschnitt.

KOMPIILERUNGSFLUSSSTEUERUNG

Fig. 2 veranschaulicht die Hauptkomponenten des OOC 100 zusammen mit dem Steuerfluß zur Kompilierung eines Abschnitts des Originalcodes. Die Haupt-OOC-Stufen sind wie folgt. Zuerst profiliert der Interpretierer 110 Verzweigungsinformationen durch die Kommunikation mit dem Verzweigungsprotokollierer 112. Der Verzweigungsprotokollierer 112 verwendet dann ein Startparameter-Auswahlverfahren, um zu bestimmen, welche Startparameter an den Kompilierer 104 zu senden sind. Anschließend verwendet der Blockwähler 114 den Startparameter und die Verzweigungsprofilinformationen, um ein Segment des zu kompilierenden Originalcodes zu wählen. Dann schafft der Blockwähler 114 einen Steuerflußgraphen (CFG), der die zu kompilierenden Originalinstruktionen beschreibt, und reicht den CFG an die Blockaufbaueinheit 116 weiter.

Anschließend flacht die Blockaufbaueinheit 116 den Steuerflußgraphen zu einer linearen Liste von Instruktionen ab. Die optimierende Codegenerierungseinheit 118 führt die tatsächliche Kompilierung der Originalinstruktionen in übersetzte Codesegmentinstruktionen durch. Der produzierte übersetzte Code, zusammen mit Informationen über das Segment, das kompiliert wird, wird schließlich zur Segmentinstallationseinheit 120 weitergereicht, die den Code für den Interpretierer 110 verfügbar macht.

OOC-AUSFÜHRUNGSSTEUERFLUSS

Fig. 3 veranschaulicht den Steuerfluß im OOC während der normalen Ausführung. Während der Interpretierer 110 den Code ausführt, kann der OOC in den Verzweigungsprotokollierer 112 einspringen, wenn er bestimmte Instruktionen ausführt. Der Verzweigungsprotokollierer 112 kann entweder zum Interpretierer 110 zurückspringen, oder, wenn das Ziel der Verzweigung bereits kompiliert wurde, in eines der installierten Segmente des kompilierten Codes einspringen. Vom kompilierten Code können Übergänge von Segment zu Segment oder zurück zum Interpretierer 110 durchgeführt werden. Der kompilierte Code kann entweder den Interpretierer 110 aufrufen, um eine einzelne Originalinstruktion auszuführen, oder kann zum Interpretierer 110 springen, wobei die gesamte Steuerung an den Interpretierer 110 weitergereicht wird.

Eine Beschreibung der ersten Ausführungsform der vorliegenden Anmeldung wird wie folgt unterteilt. Der erste Abschnitt beschreibt die Schnittstelle zwischen dem Interpretierer 110 und dem Kompilierer 104. Der zweite Abschnitt beschreibt die Modifikationen, die am Interpretierer 110 für den OOC durchgeführt wurden. Der dritte Abschnitt beschreibt den Kompilierer 104. Der abschließende Abschnitt beschreibt eine Windows-Testumgebung.

Der Beschreibung der ersten Ausführungsform folgt eine Beschreibung der zweiten bis neunten Ausführungsform der vorliegenden Erfindung.

IV. KOMMUNIKATION (GEMEINSAME EINHEIT)

Der Interpretierer 110 und der Kompilierer 104 kommunizieren miteinander auf verschiedenen Wegen. Der Interpretierer 110 zeichnet Verzweigungsinformationen in ein Verzweigungsprotokoll auf, indem er mit dem Verzweigungsprotokollierer 112 kommuniziert. Der Kompilierer 104 kann auch das Verzweigungsprotokoll lesen. Der Kompilierer 104 schafft kompilierte Codesegmente und speichert ihre Einsprungpunkte in der Übersetzungstabelle, die der Interpretierer 110 liest. Der Interpretierer 110 sendet auch Startparameteradressen an den Kompilierer 104 über einen Puffer. Der Quellencode, der sowohl vom Kompilierer 104 als auch vom Interpretierer 110 für diese Kommunikation verwendet wird, befindet sich im gemeinsamen Verzeichnis. Dieser Abschnitt beschreibt, wie die Kommunikation funktioniert.

GEMEINSAM GENUZTER OOC-PUFFER

Die gesamte Kommunikation zwischen dem Kompilierer 104 und dem Interpretierer 110 wird durch den OOC-Pufferbereich geführt, der eine große Region des gemeinsam genutzten Speichers ist. Bestimmte Kommunikation verwendet auch Systemaufrufe, um Nachrichten vom Interpretierer 110 an den Kompilierer 104 und zurück zu senden.

Die nachstehende Tabelle 1 veranschaulicht eine Abbildung der statisch zugeordneten Teile des OOC-Puffers. Der Rest des Puffers wird dynamisch für verschiedene Datenstrukturen zugeordnet, die in der auch nachstehend angegebenen Tabelle 2 gezeigt werden. Einige Felder im statisch zugeordneten Teil des OOC-Puffers zeigen auf Datenstrukturen im dynamisch zugeordneten Teil. Diese Zeiger haben hochgestellte Nummern, um darzustellen, wohin sie zeigen. Das Zonenfeld im statisch zugeordneten Teil hat beispielsweise die Nummer 2, und das Zonenfeld zeigt auf die Zonenspeicher-Datenstruktur im dynamisch zugeordneten Teil, die auch die Nummer 2 hat.

Tabelle 1

Der statisch zugeordnete Teil des OOCT-Puffers

Feld	Di- stanz	Inhalt
jump_table	0h	Ein Array von Einsprungpunkten im Interpretierer 110, wie IC_FETCH02, IU_PGMxx. OOCT_INIT schreibt sie, und der Kompilierer 104 liest sie. Der Kompilierer 104 verwendet sie, um Sprünge zum Interpretierer 110 zu generieren.
trans_master_ target_ table ¹	1000h	Ein Array von Zeigern, einer für jede Seite im ASP-Adressenraum. Für eine Seite, die der ASP nicht verwendet, ist der Zeiger 0. Für eine Seite, die der ASP verwendet, zeigt der Zeiger auf ein Array im dynamisch zugeordneten Teil des OOCT-Puffers (siehe unten).
unallocated	41004h	Ein Zeiger, der auf das erste ungenutzte Byte im dynamisch zugeordneten Teil des Puffers zeigt. Wird nur während der Initialisierung verwendet.
length_left	41008h	Die Anzahl von Bytes, die im dynamisch zugeordneten Teil des Puffers zurückbleiben. Wird nur während der Initialisierung verwendet.
num_exces	4100Ch	Nummer des Interpretierers 110.

Feld	Di- stanz	Inhalt
zones ²	41010h	Ein Zeiger auf den Zonenspeicher, der im dynamisch zugeordneten Teil des OOC- T -Puffers ist. OOC_INIT schreibt den Zeiger, während der Kompilierer 104 den Zeiger liest. Der Kompilierer 104 verwendet den Zonenspeicher während der Kompilierung.
zones_length	41014h	Der Betrag des Zonenspeichers. Wird von OOC_INIT geschrieben und vom Kompilierer 104 gelesen.
segments ³	41018h	Ein Zeiger auf den Segmentspeicher, der im dynamisch zugeordneten Teil des OOC-Puffers ist. OOC_INIT schreibt den Zeiger, während der Kompilierer 104 den Zeiger liest. Der Kompilierer 104 verwendet den Segmentspeicher, um den kompilierten Code zu speichern.
segments_ length	4101Ch	Der Betrag des Segmentspeichers. Wird von OOC_INIT geschrieben und vom Kompilierer 104 gelesen.
branch_ll_ tables ⁴	41020h	Ein Zeiger auf Verzweigungscache-strukturen erster Ordnung (L1), die im dynamisch zugeordneten Teil des OOC-Puffers sind.
branch_record_ free_ list ⁵	41024h	Eine Liste ungenutzter BRANCH_RECORD Strukturen, die im dynamisch zugeordneten Teil des OOC-Puffers sind.

Feld	Di- stanz	Inhalt
branch_header_ table ⁵	41028h	Eine Hash-Tabelle, die BRANCH_RECORD-Strukturen enthält. Die Tabelle wird dynamisch im OOCT- Puffer zugeordnet.
branch_log_ lock	4102Ch	Eine Verriegelung, die gehalten werden muß, um das Verzweigungspro- tokoll zu schreiben.
branch_seed_ buffer	41030h	Ein Puffer, den der Interpretierer 110 verwendet, um Startparameter an den Kompilierer 104 zu senden.
num_monitor_ seed_ messages	41060h	Ein Zähler, der mitteilt, wie viele Nachrichten der Interpretierer 110 an den Kompilierer 104 gesendet hat, die der Kompilierer 104 jedoch nicht beendet hat.
seed_ threshold_mode	41064h	Eine Flagge, die dem Interpretierer 110 mitteilt, wie ein Startparameter zu wählen ist. Der Startparameter ist entweder OOCT_DEBUG_MODE oder OOCT_PERFORMANCE_MODE.
seed_ production_ threshold	41068h	Die Schwellenanzahl von Malen, die eine Verzweigung ausführen muß, bevor ihr Ziel ein Startparameter für den Kompilierer 104 wird.
trickle_flush_ ll_rate	4106Ch	Die Anzahl von Malen, die eine Ver- zweigung in einem Ll-Cache aktuali- siert werden kann, bevor die Ver- zweigung aus dem Cache geräumt wird und in den Speicher zurückgeschrie- ben wird.
seeds_sent	41070h	frei bzw. engl. UNUSED
seeds_handled	41074h	UNUSED

Feld	Di- stanz	Inhalt
exit	41078h	Der Kompilierer 104 verwendet diese Flagge, um den Interpretierer 110 mitzuteilen, daß der Kompilierer 104 nach dem Empfang eines Signals den Betrieb eingestellt hat.
segment_exit	4107Ch	Ein Einsprungpunkt im Interpretierer 110, zu dem der kompilierte Code beim Ausgang springt. Der Code an diesem Einsprungpunkt gibt Verriegelungen frei, wenn notwendig.
segment_exit_ interp	41080h	Ein Einsprungpunkt im Interpretierer 110, zu dem der kompilierte Code beim Beenden einer Instruktion, die interpretiert werden muß, springt. Der Code an diesem Einsprungpunkt gibt Verriegelungen frei, wenn notwendig.
segment_exit_ log	41084h	Ein Einsprungpunkt im Interpretierer 110, zu dem der kompilierte Code beim Beenden einer nicht-festgelegten Verzweigungsinstruktion springt. Der Code an diesem Einsprungpunkt gibt Verriegelungen frei, wenn notwendig.
sbe_impl	41088h	Ein Einsprungpunkt im Interpretierer 110, den der kompilierte Code aufruft, um die SBE-Instruktion auszuführen.
cc_impl	4108Ch	Ein Einsprungpunkt im Interpretierer 110, den der kompilierte Code aufruft, um die CC-Instruktion auszuführen.

Feld	Di- stanz	Inhalt
mv_impl	41090h	Ein Einsprungpunkt im Interpretierer 110, den der kompilierte Code aufruft, um die MV-Instruktion auszuführen.
mv_impl_same_size	41094h	Ein Einsprungpunkt im Interpretierer 110, den der kompilierte Code aufruft, um die MV-Instruktion auszuführen, wenn die Längen beider Zeichenfolgen gleich sind.
segment_lock_mousetrap	41098h	Ein Einsprungpunkt im Interpretierer 110, den der kompilierte Code aufruft, um zu verifizieren, daß er weiterhin eine Verriegelung hält. DIES WIRD NUR ZUR DIAGNOSE VERWENDET.
breakpoint_trap	4109Ch	Ein Einsprungpunkt im Interpretierer 110, den der kompilierte Code aufruft, um das Diagnoseprogramm zu stoppen. DIES WIRD NUR ZUR DIAGNOSE VERWENDET.
segment_gates	410A0h	Ein Array von SEGMENT_GATE-Strukturen. Die SEGMENT_GATES werden zur Verriegelung von Segmenten des kompilierten Codes verwendet.
gate_free_list	710A0h	Eine Liste aktuell ungenutzter SEGMENT_GATES.
ooc_stack_bottom ⁷	710A4h	Die unterste Adresse des Stapels des Kompilierers 104. Zeigt in den dynamisch zugeordneten Teil des OOC-Puffers.

Feld	Di- stanz	Inhalt
ooc_t_stack_ top ⁷	710A8h	Die höchste Adresse des Stapels des Kompilierers 104. Zeigt in den dynamisch zugeordneten Teil des OOC-T-Puffers.
build_options	710ACh	Die für den Aufbau des Interpretierers 110 verwendeten Optionen. In ooc_t_compiler_start prüft der Kompilierer 104, daß er mit denselben Optionen aufgebaut wurde.
code_zone ²	710B0h	Ein Zeiger auf einen Bereich des dynamisch zugeordneten Speichers. Der Kompilierer 104 verwendet diesen Speicher zur temporären Schaffung eines Arrays von Zielinstruktionen. Am Ende der Kompilierung wird dieses Array in den Segmentspeicherbereich kopiert und dann gelöscht.

Im dynamisch zugeordneten Teil des OOC-T-Puffers sind die Größen von Datenstrukturen von verschiedenen Variablen abhängig. Eine ist die Anzahl von Systemseiten, die vom Betriebssystem für den Originalprozessor, wie den ASP von Fujitsu, verwendet werden. Für jede Seite eines ASP-Adressenraums, der zu übersetzende Instruktionen enthält, gibt es eine übersetzte Seite in der Übersetzungstabelle. Eine weitere Variable ist die Anzahl von Verzweigungsstrukturen, die das System zu protokollieren erwartet. Aktuell erwartet es 2^{20} Verzweigungen, was die Größe des BRANCH_RECORD-Arrays und die Verzweigungs-Anfangsblocktabelle beeinflusst. Die Anzahl der Interpretierer 110 beeinflusst die Größe des L1-Verzweigungsprotokollierer-Cache, da es für jede Aufgabe einen Cache gibt.

Fig. 4 veranschaulicht eine Abbildung des OOC-T-Puffers für eine Einstellung der Variablen. In Fig. 4 beträgt die Anzahl von ASP-Seiten 10 MB ASP-Instruktionen, die Anzahl der Interpretierer ist 4, und die Gesamtgröße des OOC-T-Puffers beträgt 128 MB.

Der dynamisch zugeordnete Teil des OOC-T-Puffers

Name	Inhalt
Translation Table ¹	Für jede Seite Adressenraum, der vom ASP genutzt wird, gibt es eine in der Übersetzungstabelle zugeordnete 16 KB-Seite. SIZE = Anzahl von Systemseiten * 16 KB.
BRANCH_RECORD array ³	Wir haben geschätzt, wie viele Verzweigungsanweisungen im ASP auftreten (derzeitige Schätzung 2 ²⁰), und ein BRANCH_RECORD wird für jede zugeordnet. SIZE = 2 ²⁰ * 24 Bytes = 24 MB.
Branch header table ⁶	Es gibt einen Zeiger auf einen BRANCH_RECORD für jede geschätzte Verzweigung. SIZE = 2 ²⁰ * 4 Bytes = 4 MB.
Branch L1 caches ⁴	Für jeden Interpretierer 110 gibt es einen Cache mit 32 Sätzen, 4 BRANCH_L1_RECORDs pro Satz. SIZE = Anzahl Ausführungen * 32 * 4 * 24 Bytes. Maximale SIZE = 16 * 32 * 4 * 24 Bytes = 49152 Bytes.
OOC stack ⁷	Ein 1 MB-Stapel.
Zone memory ²	Ein Prozentsatz des verbleibenden Speichers wird für den Zonenspeicher verwendet. Aktuell werden 50 % Speicher verwendet.
Segment memory ³	Ein Prozentsatz des verbleibenden Speichers wird für den Segmentspeicher verwendet. Aktuell werden 50 % Speicher verwendet.

VERZWEIGUNGSPROTOKOLL (VERZWEIGUNGSPROTOKOLLIERER 112)

Die Verzweigungsprotokoll-Datenstrukturen sind das BRANCH_RECORD-Array, die Verzweigungs-Anfangsblocktabelle und die Verzweigungs-L1-Caches. Für eine Erläuterung, wie der Verzweigungsprotokollierer 112 arbeitet, siehe bitte nachstehend angegebener Abschnitt über Interpretierermodifikationen. Dieser Abschnitt beschreibt, wie das Verzweigungsprotokoll verwendet wird, um Informationen vom Interpretierer 110 zum Kompilierer 104 zu kommunizieren.

Fig. 4 veranschaulicht den OOC-T-Puffer nach der Initialisierung. Die Größen der Regionen sind maßstabgetreu gezeichnet. Für dieses Beispiel beträgt die Größe des OOC-T-Puffers 128 MB, die Anzahl von ASP-Seiten beträgt 2560, die Anzahl von Interpretierern 110 beträgt 2, und die erwartete Anzahl von Verzweigungsanweisungen beträgt 220.

Der Kompilierer 104 liest das Verzweigungsprotokoll, um herauszufinden, wie viele Male eine bedingte Verzweigungsanweisung eingeschlagen wurde, und wie viele Male eine bedingte Verzweigungsanweisung nicht eingeschlagen wurde. Der Kompilierer 104 verwendet diese Informationen auf zwei Wegen. Erstens, wenn der Kompilierer 104 die In-

struktionen einer Sprachanalyse. Er zieht, versucht der Kompilierer 104 nur die Instruktionen der Sprachanalyse zu unterziehen, die am häufigsten ausgeführt wurden. Wenn eine bedingte Verzweigungsinstruktion auftritt, prüft er, wie viele Male sie sich verzweigt hat, und wie viele Male sie fehlgeschlagen ist. Zweitens, wenn der Kompilierer 104 einen Code generiert, versucht der Kompilierer, die wahrscheinlichste Nachfolgerinstruktion einer bedingten Verzweigung unmittelbar nach der Verzweigungsinstruktion zu platzieren. Dadurch läuft der generierte Code schneller. Um mitzuteilen, welcher Nachfolger wahrscheinlicher ist, verwendet der Kompilierer 104 Verzweigungsprotokollinformationen. Für nähere Einzelheiten siehe bitte nachstehende Informationen über den Kompilierer 104.

BRANCH_Get_Record (ooc/compiler/branch.c)

Wenn der Kompilierer 104 Verzweigungsprotokollinformationen lesen will, ruft er die Prozedur BRANCH_Get_Record mit der Adresse der Verzweigungsinstruktion auf. Diese Prozedur schlägt die Verzweigung im Verzweigungsprotokoll nach und führt einen Zeiger auf eines der Elemente des BRANCH_RECORD-Arrays zurück. Dann kann der Kompilierer 104 sehen, wie viele Male die Verzweigungsinstruktion ausgeführt wurde, wie viele Male sie sich verzweigt hat, und wie viele Male sie fehlgeschlagen ist.

ÜBERSETZUNGSTABELLE (ÜBERSETZUNGSEINHEIT)

Die Übersetzungstabelle enthält Informationen über jede Instruktion im ASP-Adressenraum. Die Übersetzungstabelle zeichnet auf, ob die Instruktion das Ziel einer Verzweigung ist (JOIN), ob die Instruktion zum Kompilierer 104 als Startparameter gesendet wurde (BUFFERED) und ob es einen kompilierten Einsprungpunkt für das Segment gibt (ENTRY). Wenn der OOC initialisiert wird, ist die Übersetzungstabelle leer. Wenn die Verzweigungsinstruktionen protokolliert werden, werden ihre Ziele als JOIN-Punkte markiert. Wenn die Verzweigung mehrere Male als die Schwelle ausführt, wird das Ziel als Startparameter an den Kompilierer 104 gesendet, und der Übersetzungstabelleneintrag wird als BUFFERED markiert. Nachdem der Kompilierer 104 die Kompilierung der übersetzten Version beendet, speichert er die Adressen von Einsprungpunkten in der Übersetzungstabelle und markiert sie als ENTRYs.

Fig. 5a, 5b und 5c veranschaulichen die Struktur einer Übersetzungstabelle gemäß einer bevorzugten Ausführungsform der vorliegenden Erfindung. Wie in Fig. 5a veranschaulicht, wird eine ASP-Adresse in zwei Teile geteilt. Die oberen 20 Bits sind die Seitennummer, und die unteren 12 Bits sind die Seitendistanz.

Fig. 5b veranschaulicht, daß die Seitennummer als Index in die Übersetzungstabelle erster Ordnung verwendet wird. Die Seiten, die der ASP bearbeitet, sind in der Tabelle erster Ordnung. Die Seiten, die der ASP nicht verwendet, haben keine Zeiger, da es niemals eine Instruktion mit dieser Seitennummer geben wird. Die Zeiger zeigen in die Übersetzungstabelle zweiter Ordnung. Das Hinzufügen einer Seitendistanz zum Zeiger ergibt einen Übersetzungstabelleneintrag.

Wie in Fig. 5c veranschaulicht, ist jeder Eintrag 32 Bits lang, und seine Felder sind unten dargestellt. Das erste Bit besagt, ob die ASP-Instruktion ein Verbindungspunkt ist. Das zweite besagt, ob es einen Segmenteinsprungpunkt für die Instruktion gibt. Das dritte besagt, ob die Instruktion zum Kompilierer 104 als Startparameter gesendet wurde. Die anderen Bits des Übersetzungstabelleneintrags sind die Einsprungpunktadresse für die Instruktion, wenn es eins gibt, oder 0 gibt, falls kein Einsprungpunkt vorliegt.

Da die K-Maschinenarchitektur Instruktionen mit variabler Länge aufweist, hat die Übersetzungstabelle einen Eintrag für jede ASP-Adresse, einschließlich der Adressen, die in der Mitte der Instruktionen und Datenadressen liegen. Dies macht die Tabelle sehr groß, es vereinfacht jedoch das Vorhaben der Lokalisierung des Übersetzungstabelleneintrags für eine Adresse. Die Struktur der Übersetzungstabelle ist in Fig. 5a, 5b und 5c gezeigt. Wie oben angegeben, hat die Übersetzungstabelle zweiter Ordnung einen 32 Bit-Eintrag für jede ASP-Adresse. Wenn daher der ASP 10 MB Raum verwendet, verwendet die Übersetzungstabelle zweiter Ordnung 40 MB. Es gibt verschiedene Prozeduren und Makros, welche die Einträge der Übersetzungstabelle lesen und schreiben.

TRANS_Set_Entry_Flag (ooc/common/trcommon.h)

Das Makro TRANS_Set_Entry_Flag schaltet eine der Flaggen JOIN, ENTRY oder BUFFERED des Übersetzungstabelleneintrags ein. Es verwendet eine Assemblersprachinstruktion mit dem Verriegelungspräfix, so daß es das Bit automatisch setzt.

TRANS_Reset_Entry_Flag (ooc/common/trcommon.h)

Das Makro TRANS_Reset_Entry_Flag schaltet eine der Flaggen JOIN, ENTRY oder BUFFERED des Übersetzungstabelleneintrags aus. Es verwendet eine Assemblersprachinstruktion mit dem Verriegelungspräfix, so daß es das Bit automatisch zurücksetzt.

TRANS_Entry_FlagP (ooc/common/trcommon.h)

Das Makro TRANS_Entry_FlagP liest den Zustand einer der Flaggen JOIN, ENTRY oder BUFFERED des Übersetzungstabelleneintrags und führt ihn zurück.

TRANS_Test_And_Set_Entry_Flag (ooc/common/trcommon.h)

Die Prozedur TRANS_Test_And_Set_Entry_Flag liest automatisch den Zustand einer der Flaggen JOIN, ENTRY oder BUFFERED und schaltet sie ein, wenn sie nicht bereits eingeschaltet war. Sie führt den Zustand der Flagge zurück, bevor sie die Prozedur aufruft.

TRANS_Set_Entry_Address (ooct/common/trcommon.h)

Die Prozedur TRANS_Set_Entry_Address schreibt die Einsprungpunktadresse des Übersetzungstabelleneintrags. Sie verwendet eine Assemblersprachinstruktion mit dem Verriegelungspräfix, so daß sie die Adresse automatisch schreibt.

5 Es ist zu beachten, daß eine Einsprungpunktadresse die Adresse einer Zielinstruktion ist, wenn keine Segmentverriegelung besteht, jedoch die Adresse einer SEGMENT_GATE-Datenstruktur ist, wenn eine Segmentverriegelung besteht.

TRANS_Get_Entry_Address (ooct/common/trcommon.h)

10 Die Prozedur TRANS_Get_Entry_Address liest die Einsprungpunktadresse des Übersetzungstabelleneintrags und führt diese zurück. Es ist zu beachten, daß eine Einsprungpunktadresse die Adresse einer Zielinstruktion ist, wenn keine Segmentverriegelung besteht, jedoch die Adresse einer SEGMENT_GATE-Datenstruktur ist, wenn eine Segmentverriegelung besteht.

SEGMENTE

Ein Segment ist eine Einheit eines kompilierten Codes, der vom KOI-System ausgeführt werden kann. Nachstehend angegebenes Material über den Kompilierer 104 beschreibt, wie der Kompilierer 104 den Interpretierer 110 über ein Segment informiert, wie der Interpretierer 110 in das Segment einspringt und dieses verläßt, und wie der Kompilierer 104 den Interpretierer 110 anweist, die Verwendung eines Segments zu stoppen und zu einem anderen zu wechseln.

20 Wenn ein Segment geschaffen wird, gibt es einige ASP-Instruktionsadressen, wo der Interpretierer 110 in das Segment einspringen kann. Für jede dieser Adressen schafft der Kompilierer 104 einen Einsprungpunkt in das Segment. Ein Einsprungpunkt ist ein spezieller Punkt im Segment, wo der Interpretierer 110 hinspringen darf. An anderen Punkten im Segment nimmt der kompilierte Code an, daß bestimmte Werte in Registern vorliegen, so daß es nicht sicher ist, dorthin zu springen. Um dem Interpretierer 110 mitzuteilen, wo die Einsprungpunkte sind, ruft der Kompilierer 104 TRANS_Set_Entry_Address für jede n.te TRANS_Get_Entry_Address auf.

25 Der Interpretierer 110 prüft auf kompilierte Codesegmente, wenn sie in den Verzweigungsprotokollierer 112 einspringen. Sie rufen TRANS_Entry_FlagP auf, um zu sehen, ob die aktuelle ASP-Adresse einen Einsprungpunkt aufweist. Wenn sie dies tut, rufen sie TRANS_Get_Entry_Address auf, um die Adresse zu lesen. Wenn die Segmentverriegelung ein ist, verriegeln sie das Segment (siehe unten) und springen dann zum Einsprungpunkt. Wenn die Segmentverriegelung aus ist, springen sie nur zum Einsprungpunkt. Der kompilierte Code entscheidet, wenn er aussteigen soll. Dies geschieht üblicherweise, wenn er eine Instruktion ausführen muß, die nicht Teil desselben Segments ist, so springt er zum Interpretierer 110.

30 Der Kompilierer 104 kann ein kompiliertes Codesegment löschen und den Interpretierer 110 anweisen, ein anderes zu verwenden. Der Kompilierer 104 tut dies, indem er das ENTRY-Bit des Übersetzungstabelleneintrags ausschaltet, die Einsprungpunktadresse ändert und dann das ENTRY-Bit wieder einschaltet.

SEGMENTVERRIEGELUNG

40 Die Segmentverriegelung ist ein optionales Merkmal des OOCT-Systems. Da der Verzweigungsprotokollierer 112 mit dem Laufen des Systems mehr Informationen sammelt, kann der Kompilierer 104 eine neue Version eines Segments produzieren, die besser ist als die alte. Die Segmentverriegelung ermöglicht dem Kompilierer 104, ein altes Segment durch ein neues zu ersetzen und den vom alten Segment verwendeten Speicher erneut zu beanspruchen. Leider macht die Segmentverriegelung den Verzweigungsprotokollierer 112 und den kompilierten Code langsamer. Daher kommt es zu einem

45 Abtausch zwischen der Zeit zur Ausführung des OOCT-Codes und dem Raum, den er verwendet. Dieser Abschnitt beschreibt, wie die Segmentverriegelung arbeitet.

Der Segmentverriegelungscode hat zwei Hauptteile. Der erste Teil ist eine Schnittstelle für alle Teile des OOCT-Systems mit Ausnahme der Segmentverriegelungsimplementation.

50 Diese Schnittstelle garantiert, daß sich ein Segment nur in einem von vier gut definierten Zuständen befinden kann, und Zustände auf gut definierten Wegen automatisch ändert. Der zweite Teil ist die Implementation der Segmentverriegelung selbst, wodurch die von der Schnittstelle gegebenen Garantien erfüllt werden.

AUSBILDUNG (DESIGN)

55 Die Zustände, in denen sich ein Segment befinden kann, werden in Tabelle 3 gezeigt. Ein Segment kann entweder erreichbar oder unerreichbar sein, und es kann entweder verriegelt oder unverriegelt sein. Segmente sind erreichbar, wenn es einen oder mehrere Einsprungpunkte in die Übersetzungstabelle gibt. Es ist unerreichbar, wenn es keine Einsprungpunkte in das Segment in der Übersetzungstabelle gibt. Ein Einsprungpunkt ist eine Struktur, die eine Verriegelung und eine Instruktionsadresse enthält. Die Verriegelung, die gleichzeitig von mehr als einem Interpretierer 110 verwendet werden kann, zählt, wie viele Interpretierer 110 den Einsprungpunkt und das diesen enthaltende Segment verwenden. Ein

60 Segment ist verriegelt, wenn einer oder mehrere seiner Einsprungpunkte verriegelt sind. Es ist unverriegelt, wenn alle seiner Einsprungpunkte unverriegelt sind.

Der Kompilierer 104 kann ein Segment erneut beanspruchen und löschen, wenn es unerreichbar und unverriegelt ist, er kann es jedoch nicht erneut beanspruchen, wenn es erreichbar oder verriegelt ist. Jedes Segment beginnt im Zustand U/U, wenn es der Kompilierer 104 schafft. Es bewegt sich in einen Zustand R/U, wenn der Kompilierer 104 seine Einsprungpunkte in die Übersetzungstabelle schreibt. Es kann sich zu einem Zustand R/L und zurück zu R/U bewegen, während Interpretierer 110 in das Segment einspringen und dieses verlassen. Der Kompilierer 104 kann ein neues Segment schaffen, das dieselben Instruktionen übersetzt wie ein altes Segment. In diesem Fall überschreibt er die Einsprung-

punkte des alten Segments in der Übersetzungstabelle, wodurch es unerreichbar wird. Wenn der Kompilierer 104 den letzten Eintrag des Segments überschreibt, geht es vom Zustand R/L zu U/L, wenn es von einem Interpretierer 110 verwendet wird, oder vom Zustand R/U zu U/U, wenn es von keinem Interpretierer 110 verwendet wurde. Schließlich geben alle das Segment verwendenden Interpretierer 110 ihre Verriegelungen frei, und das Segment befindet sich im Zustand U/U. Der Kompilierer 104 kann dann das Segment erneut beanspruchen und es löschen, da es von keinem Interpretierer 110 verwendet wird, und keiner in dieses springen kann.

Tabelle 3

Die Zustände, in denen sich ein Segment befinden kann

Zustand	Erreichbar	Verriegelt	Beschreibung
U/U	Nein	Nein	Kein Interpretierer 110 verwendet das Segment, und kein Interpretierer 110 kann in dieses einspringen. Der Kompilierer 104 kann es jederzeit löschen.
R/U	Ja	Nein	Kein Interpretierer 110 verwendet das Segment.
R/L	Ja	Ja	Ein oder mehrere Interpretierer 110 verwenden das Segment ...
U/L	Nein	Ja	Ein oder mehrere Interpretierer 110 verwenden das Segment ...

Fig. 6 veranschaulicht einen Interpretierer 110 zum Einspringen in ein Segment 122 und Verlassen desselben gemäß einer Ausführungsform der vorliegenden Erfindung. Das Segment 122 in der Mitte der Zeichnung ist die Codeeinheit, die vom Kompilierer 104 produziert wird. Das Segment 122 muß zu jeder Zeit, wenn es vom Interpretierer 110 verwendet wird, verriegelt sein. Demgemäß wird ein Verriegelungszähler (nicht gezeigt) inkrementiert, bevor in das Segment 122 eingesprungen wird, und der Verriegelungszähler wird dekrementiert, nachdem das Segment 122 verlassen wurde. Da der Interpretierer 110 den Einsprungpunkt nicht nachschlagen und den Einsprungpunkt automatisch verriegeln kann, muß bestimmt werden, ob der Einsprungpunkt, nachdem er verriegelt wurde, nicht geändert wurde.

Fig. 7 veranschaulicht ein Verfahren des Kompilierers 104, um ein Segment zu schaffen, das Segment durch den Interpretierer 110 erreichbar zu machen, alte Segmente unerreichbar zu machen, und alte Segmente zu löschen. In Schritt S200 schafft der Kompilierer 104 ein neues Segment und fügt assoziierte Einsprungpunkte zur Übersetzungstabelle hinzu. Wenn ein Einsprungpunkt in Schritt S200 hinzugefügt wird, kann ein älterer Einsprungpunkt umgeschrieben werden. Der ältere Einsprungpunkt ist nun unerreichbar, und kann demgemäß erneut verwendet werden, wenn keine Aufgabe (wie der Interpretierer 110 oder Kompilierer 104) eine Verriegelung darauf hält. Der alte Einsprungpunkt wird auf eine Liste zum erneuten Beanspruchen (nicht gezeigt) gesetzt.

Der Schritt 202 veranschaulicht, wie der Kompilierer 104 die Liste zum erneuten Beanspruchen verwendet. Der Schritt 202 prüft, ob ein Einsprungpunkt verriegelt ist. Wenn der Einsprungpunkt nicht verriegelt ist, dann wird der Einsprungpunkt von keinem Interpretierer 110 verwendet, und kann daher aus dem Segment, zu dem er gehört, entfernt werden. Wenn das Segment jedoch keine weiteren Einsprungpunkte hat, dann wird das Segment nicht von einer Aufgabe (wie dem Interpretierer 110 und Kompilierer 104) verwendet, und keine Aufgabe kann in dieses springen. Daher kann das Segment gelöscht werden.

Die Segmentverriegelungsschnittstelle ermöglicht, daß mehrere Teile des OOC die Details der Synchronisation ignorieren, da sich ein Segment immer in einem gut definierten Zustand zu befinden scheint, und alle Zustandsübergänge automatisch zu erfolgen scheinen. Innerhalb des Segmentverriegelungscodes erfolgen jedoch die Übergänge nicht automatisch, da das Intel-Ziel einen derartig komplizierten Betrieb in der Hardware nicht unterstützt. Daher läßt der Segmentverriegelungscodes die Übergänge automatisch erscheinen.

IMPLEMENTATION

Die Prozeduren zum Ausführen des Interpretierers 110 und Kompilierers 104 sind in Fig. 6 bzw. Fig. 7 veranschaulicht. Die beiden Prozeduren arbeiten zusammen, um sicherzustellen, daß jeder Übergang automatisch erscheint. Die Zahlenreferenzen in der folgenden Beschreibung beziehen sich auf 6 und Fig. 7.

Es gibt sechs mögliche Übergänge unter den vier Zuständen der Segmentschnittstelle, und sie fallen in vier Gruppen.

Der erste Übergang ist U/U zu R/U, wenn der Kompilierer 104 ein Segment erreicht, macht, indem er seine Einsprungpunkte in die Übersetzungstabelle schreibt (*306). Da der Kompilierer 104 die einzige Aufgabe ist, die in die Übersetzungstabelle schreiben darf, ist keine Synchronisation notwendig, um diesen Übergang automatisch zu machen. Die zweite Gruppe von Übergängen ist R/U zu U/U und der ähnliche von R/L zu U/L. Diese geschehen, wenn der Kompilierer 104 den letzten Einsprungpunkt eines Segments in der Übersetzungstabelle überschreibt (*306). Obwohl der Kompilierer 104 einen neuen Einsprungpunkt automatisch in die Übersetzungstabelle schreiben kann, kann der Interpretierer 110 einen Einsprungpunkt nicht automatisch lesen und verriegeln (*301, *302). Der Interpretierer 110 muß den Einsprungpunkt in einer Operation lesen und ihn in einer anderen Operation verriegeln. Dies führt zu einem potentiellen Problem, daß, wenn ein Interpretierer 110 einen alten Einsprungpunkt aus der Übersetzungstabelle liest, der Kompilierer 104 dann einen neuen schreibt, und der Interpretierer 110 anschließend den alten Einsprungpunkt verriegelt. In diesem Fall nimmt der Kompilierer 104 an, daß der Einsprungpunkt unerreichbar ist, aber der Interpretierer 110 kann in das Segment einspringen, was ein Fehler ist. Um dieses Problem zu verhindern, prüft der Interpretierer 110, daß die Übersetzungstabelle denselben Einsprungpunkt nach der Verriegelung enthält (*303). Wenn die Übersetzungstabelle denselben Einsprungpunkt enthält, ist es weiterhin erreichbar, und es ist sicher, in das Segment einzuspringen. Wenn die Übersetzungstabelle nicht denselben Eintrag enthält, muß der Interpretierer 110 seine Verriegelung freigeben und darf nicht in das Segment einspringen.

Die dritte Gruppe von Übergängen ist R/U zu R/L und umgekehrt von R/L zu R/U. Der erste geschieht, wenn ein Interpretierer 110 den Einsprungpunkt aus der Übersetzungstabelle liest und diesen verriegelt (*302). Der zweite geschieht, wenn der Interpretierer 110 ein Segment an seinem Ausgang verläßt (*304) und zur Entriegelungsprozedur geht (*305). Es ist wichtig, daß die Verriegelungs- und Entriegelungsanweisungen nicht selbst im Segment sind, da jedesmal, wenn das Segment entriegelt wird, der Kompilierer 104 dieses löschen kann (*3011).

Der vierte Übergang ist von U/L zu U/U. Er geschieht auch, wenn der Interpretierer 110 ein Segment verläßt (*304) und zur Entriegelungsprozedur geht (*305). Nachdem dieser Übergang eintritt, wird das Segment entriegelt, und der Kompilierer 104 führt die beiden Tests durch (*309, *3010) und löscht das Segment.

Da der Interpretierer 110 die Verriegelung auf einem Segment für einen willkürlichen Zeitraum halten kann, ist es ineffizient, den Kompilierer 104 auf eine Verriegelung warten zu lassen. Daher versucht der Kompilierer 104 nicht, Einsprungpunkte zu verriegeln, um den Interpretierer 110 daran zu hindern, diese zu verwenden. Statt dessen macht er das Segment nur unerreichbar und prüft später, ob die Verriegelung freigegeben wurde (*309). Sobald die Verriegelung freigegeben wird, kann der Einsprungpunkt frei gemacht und erneut verwendet werden.

ÜBERWACHUNGSNACHRICHTENWARTESCHLANGEN

Der Interpretierer 110 sendet Startparameteradressen an den Kompilierer 104. Es werden zwei Nachrichtenwarteschlangen verwendet, um sie zu senden. Die erste verwendet die KOI-Systemaufrufe ScMsgSnd und ScMsgRcv zum Senden und Empfangen von Startparametern. Die zweite Warteschlange verwendet einen gemeinsam genutzten Speicherbereich im OOC-T-Puffer. Der gemeinsam genutzte Bereich wird branch_Seed_Buffer genannt.

Der Grund für die Verwendung von zwei Warteschlangen ist, daß jede einen Vorteil und einen Nachteil hat. Der KOI-Systemaufruf ist für den Interpretierer 110 kostspielig in der Verwendung, so daß er nicht sehr häufig verwendet werden sollte. Der AOI-Systemaufruf ermöglicht es jedoch dem Kompilierer 104 zu blockieren, wenn keine Startparameter zu kompilieren sind. Dies ermöglicht, daß das KOI-System die CPU des Kompilierers 104 verwendet, um einige andere Arbeiten zu erledigen. Der Vorteil des gemeinsam genutzten Speicherpuffers ist, daß er sehr kostengünstig für den Interpretierer 110 ist, und der Nachteil ist, daß der Kompilierer 104 nicht blockieren kann, wenn es keine Startparameter gibt.

Durch die Verwendung beider Warteschlangen nützt der OOC-T die Vorteile beider Verfahren. Wenn der Kompilierer 104 untätig ist, ruft er ScMsgRcv auf, um zu blockieren. In diesem Fall sendet der Interpretierer 110 den nächsten Startparameter mit einem ScMsgSnd-Aufruf, um den Kompilierer 104 aufzuwecken. Wenn der Kompilierer 104 arbeitet, sendet der Interpretierer 110 Startparameter durch den branch_Seed_Buffer-Bereich, was schneller ist. Nachdem der Kompilierer 104 eine Kompilierung beendet, prüft er auf einen sch_Seed_Buffer-Bereich. Wenn es welche gibt, kompiliert er sie. Wenn er alle Startparameter beendet, ruft er erneut ScMsgRcv auf und blockiert.

V. INTERPRETIERERMODIFIKATIONEN (AUSFÜHRUNGSEINHEIT)

Die Ausbildung des OOC-T enthält drei Typen von Modifikationen am Interpretierer 110. Erstens muß der OOC-T vom Interpretierer 110 initialisiert werden. Zweitens wurde der Interpretierer 110 modifiziert, um eine Verzweigungsprotokollierung zu verwenden. Schließlich wurde der Interpretierer 110 modifiziert, um Übergänge zum und aus dem kompilierten Code zu ermöglichen. Dieses Dokument beschreibt die Details dieser Modifikationen.

Der OOC-T-Interpretercode kann in zwei Modi laufen, OOC-T_PERFORMANCE_MODE und OOC-T_DEBUG_MODE. Diese Dokumentation beschreibt alle Merkmale des OOC-T_PERFORMANCE_MODE und gibt an, wo sich der OOC-T_DEBUG_MODE davon unterscheidet.

INITIALISIERUNG

Bevor der OOC-T irgendeinen Code kompiliert oder irgendwelche Verzweigungen protokolliert, ruft der Interpretierer 110 OOC-T_INIT auf, um die OOC-T-Datenstrukturen zu initialisieren. OOC-T_INIT und die Prozeduren, die dadurch aufgerufen werden, führen die folgenden Schritte durch.

Die Übersetzungstabelle initialisieren. Die MCD-Instruktion informiert den OOC-T über die Seiten im Adressenraum des Systems. Die Prozedur TRANS_Execution_Init schafft die Übersetzungstabelle erster Ordnung so, daß die Einträge für Systemseiten auf Übersetzungstabellen-Arrays zweiter Ordnung zeigen. Diese Arrays werden bei der Initialisierung auf Null gesetzt. Für genauere Details über die Übersetzungstabelle siehe Kommunikationsabschnitt.

Verzweigungsprotokollierer 112 initialisieren. Die Prozedur `BRANCH_Execution_Init` initialisiert den Speicher im `OOCT_buffer` für einige Datenstrukturen. Erstens gibt es das Verzweigungsprotokoll selbst, das Profilinformatoren über Verzweigungsstrukturen enthält. Zweitens gibt es einen Cache erster Ordnung (L1), der den Verzweigungsprotokollierer 112 schneller operieren läßt. Drittens gibt es einen Startparameterpuffer, der vom Verzweigungsprotokollierer 112 zum Kompilierer 104 gesendete Startparameter enthält. Viertens gibt es einige globale Funktionen, die der kompilierte Code aufruft. Ihre Adressen werden im `OOCT_buffer` während `BRANCH_Execution_Init` gespeichert. Für nähere Informationen über das Verzweigungsprotokoll und den Cache erster Ordnung siehe obiger Abschnitt über den Verzweigungsprotokollierer 112.

Stapelspeicher des Kompilierers 104 zuordnen. Der Kompilierer 104 verwendet einen speziellen großen Stapel, der im `OOCT_buffer` zugeordnet wird.

1. Zonenspeicher des Kompilierers 104 zuordnen. Der Kompilierer 104 verwendet diesen Speicher im `OOCT_buffer` während der Kompilierung.
2. Den kompilierten Segmentspeicher zuordnen. Der kompilierte Code wird in diesem Bereich des `OOCT_buffer` platziert.
3. Statistische Informationen auf Null setzen. Die meisten Informationen im `OOCT`-Statistikbereich werden zurückgesetzt, wenn der `OOCT` initialisiert wird.

VERZWEIGUNGSPROTOKOLLIERER

SCHNITTSTELLE ZUM INTERPRETIERER

Wenn der Interpretierer 110 eine Verzweigungsstrukturen im Systemcode ausführt, und das `OOCT`-Modusbit gesetzt ist, ruft der Interpretierer 110 den Verzweigungsprotokollierer 112 durch eine der folgenden Routinen auf:

`_declspec(naked) OOCT_Log_Unconditional_Fixed_Branch()`

Abgerufen vom Interpretierer mit einer Verzweigung

Argumente: ausf.: Adresse der Verzweigungsinstruktion

Rücksprünge: Springt nicht zurück (verhält sich wie ein Sprung zu IC_FETCH02)

`_declspec(naked) OOCT_Log_Unconditional_Non_Fixed_Branch()`

Abgerufen vom Interpretierer mit einer Verzweigung

Argumente: ausf.: Adresse der Verzweigungsinstruktion

Rücksprünge: Springt nicht zurück (verhält sich wie ein Sprung zu IC_FETCH02)

`_declspec(naked) OOCT_Log_Conditional_Fixed_Branch_Taken()`

Abgerufen vom Interpretierer mit einer Verzweigung

Argumente: ausf.: Adresse der Verzweigungsinstruktion

Rücksprünge: Springt nicht zurück (verhält sich wie ein Sprung zu IC_FETCH02)

`_declspec(naked) OOCT_Log_Conditional_Fixed_Branch_Not_Taken()`

Abgerufen vom Interpretierer mit einer Verzweigung

Argumente: ausf.: Adresse der Verzweigungsinstruktion

Rücksprünge: Springt nicht zurück (verhält sich wie ein Sprung zu IC_FETCH02)

Diese vier Routinen prüfen auf einen kompilierten Einsprungpunkt für die Zieladresse und springen zum Einsprungpunkt, wenn er existiert. Wenn er nicht existiert, darin aktualisieren die Routinen das Verzweigungsprotokoll durch das Aufrufen von `branch_L1_Touch` (siehe nächster Abschnitt), und springen dann zur Abholroutine des Interpretierers 110.

AKTUALISIERUNG VON VERZWEIGUNGSPROTOKOLLTABELLEN

Fig. 8 veranschaulicht eine Struktur eines `BRANCH_RECORD` gemäß einer bevorzugten Ausführungsform der vorliegenden Erfindung.

Der Verzweigungsprotokolliercode zählt, wie viele Male eine Verzweigung ausgeführt hat. Es gibt zwei Datenstrukturen, die der Verzweigungsprotokollierer 112 zum Speichern der Zählungen verwendet. Erstens gibt es das Verzweigungsprotokoll, das von allen simulierten Prozessoren in einem Mehrprozessorsystem gemeinsam genutzt wird. Zweitens gibt es einen Cache erster Ordnung (L1) für jeden simulierten Prozessor im System. Die Verzweigungsausführungs-

zählungen werden zuerst in den **Fig. 9** geschrieben und dann in das Verzweigungsprotokoll geschrieben. Dieser Abschnitt beschreibt die Struktur der L1-Caches und des Verzweigungsprotokolls. Er beschreibt auch, wie der Verzweigungsprotokollierer **112** diese verwendet.

Die Informationen für jede Verzweigung werden in einer Struktur gespeichert, die **BRANCH_RECORD** genannt wird. Sie enthält die Adresse der Verzweigung, das Ziel der Verzweigung, die Fehlschlag-Instruktion nach der Verzweigung, die ungefähre Anzahl von Malen, welche die Verzweigung ausgeführt hat, und die ungefähre Anzahl von Malen, welche die Verzweigung eingeschlagen wurde. Das letzte Feld des **BRANCH_RECORD** ist ein Zeiger auf einen anderen **BRANCH_RECORD**. Er wird verwendet, um den **BRANCH_RECORD** in einer verketteten oder verknüpften Liste zu verbinden.

Die Hash-Tabelle ist als Array verketteter Listen organisiert.

Fig. 9 veranschaulicht die Struktur des Verzweigungsprotokolls. Es ist eine große Hash-Tabelle, die **BRANCH_RECORDS** speichert. Jeder Interpretierer **110** hat seine eigene Kopie der variablen **local_branch_header_table**, sie zeigen jedoch alle auf dasselbe Array im OOC-T-Pufferbereich. Die Elemente der **local_branch_header_table** sind Zeiger auf Listen von **BRANCH_RECORDS**. Die Prozedur zum Finden eines **BRANCH_RECORD** für eine Verzweigung weist 3 Schritte auf.

1. Hash-Codieren der Zieladresse. ($\text{index} = \text{BRANCH_HASH}(\text{destination_address}) \% \text{BRANCH_TABLE_SIZE}$.)
2. Kopf der Liste bzw. engl. head of list ermitteln. ($\text{list} = \text{local_branch_header_table}[\text{index}]$.)
3. Liste abwärts durchgehen, bis ein Datensatz mit derselben Verzweigungsadresse gefunden wird. ($\text{while} (\text{list} \rightarrow \text{branch_address} \neq \text{branch_address}) \text{list} = \text{list} \rightarrow \text{next}$.)

Fig. 9 veranschaulicht insbesondere, daß die variable **local_branch_header_table** ein Array von Zeigern auf Listen ist. Jede Liste enthält **BRANCH_RECORDS**, die dieselbe Zieladresse aufweisen. Wenn es keine Liste gibt, ist der Zeiger in der **local_branch_header_table** NULL.

Das Verzweigungsprotokoll enthält alle Informationen über Verzweigungen, es weist jedoch auch zwei Probleme auf. Erstens sind das Nachschlagen und Einfügen von **BRANCH_RECORDS** langsame Operationen. Sie sind zu langsam durchzuführen, jedesmal wenn der Interpretierer **110** eine Verzweigung protokolliert. Zweitens verwendet jeder Interpretierer **110** dasselbe Verzweigungsprotokoll. Um die Listen der **BRANCH_RECORDS** konsistent zu halten, kann nur eine Ausführung zu einer Zeit auf das Verzweigungsprotokoll zugreifen. Dies verlangsamt das Mehrprozessorsystem noch mehr als das Einprozessorsystem. Um diese Probleme zu beheben, gibt es einen L1-Cache für jeden Interpretierer **110**. Auf den L1-Cache kann rasch zugegriffen werden, und die Interpretierer **110** können auf ihre L1-Caches parallel zugreifen. Jeder L1-Cache ist ein zweidimensionales Array von **BRANCH_L1_RECORD**-Strukturen. Die Basisadresse des Arrays wird in der variablen **branch_L1_table** gespeichert.

Fig. 10 veranschaulicht die Struktur des L1-Cache. Der Cache ist ein zweidimensionales Array von **BRANCH_L1_RECORDS**. Die erste Dimension ist **BRANCH_L1_SETS** (aktuell 32), und die zweite Dimension ist **BRANCH_L1_SETSIZE** (aktuell 4). Jede Reihe des Arrays ist ein Satz. Dieselbe Verzweigungsinstruktion verwendet immer denselben Satz des Cache, er kann jedoch an verschiedenen Plätzen sein.

Wie in **Fig. 10** veranschaulicht, ist der L1-Cache in Sätzen organisiert. Die Satznummer für eine Verzweigung ist gleich $(\text{branch_address} + \text{branch_destination}) \% \text{BRANCH_L1_SETS}$. Die 4 Mitglieder des Satzes halten die 4 neuesten Verzweigungen mit derselben Satznummer. Dies wird 4-weg-Satzassoziativität genannt. Sie verbessert die Leistung des Cache, wenn einige Verzweigungen nahezu zur gleichen Zeit ausgeführt werden, die dieselbe Satznummer aufweisen.

Fig. 11 veranschaulicht ein Verfahren zur Ausführung des Betriebs des L1-Cache durch den Interpretierer **110** gemäß einer Ausführungsform der vorliegenden Erfindung. Mit anderen Worten veranschaulicht **Fig. 11** ein Verzweigungsprotokollverfahren unter Verwendung des L1-Cache.

Das optimierende Objektcode-Übersetzungsverfahren verwendet zwei Speicherformen zum Aufzeichnen nicht-kompilierter Verzweigungen, nämlich

1. ein Verzweigungsprotokoll mit einer sich dynamisch ändernden Größe proportional zur Anzahl aufgezeichneter Verzweigungen, und
2. einen Verzweigungs-Cache, der als L1-Cache bezeichnet wird, in dem eine begrenzte Anzahl nicht-kompilierter aufgezeichneter Verzweigungen in einer Reihenfolge gespeichert werden, die den Zugriff erweitert.

Das Verzweigungsprotokoll und der L1-Cache repräsentieren virtuelle Speicherstellen, die von einem Betriebssystem verwaltet werden. Daher wird die Bezeichnung "L1-Cache" willkürlich dem Cache zum Speichern nicht-kompilierter Verzweigungen verliehen und sollte nicht mit dem "L1-Cache" verwechselt werden, der allgemein auf einem Prozessor wie dem Pentium Pro zu finden ist.

Der optimierende Objektcode-Übersetzer gemäß der vorliegenden Erfindung sieht vor, daß der Interpretierer **110** eine Vielzahl verschiedener Verzweigungsprotokollerroutinen aufrufen kann. Jede Verzweigungsprotokollerroutine selbst ruft jedoch eine Subroutine auf, die entscheidet, zum kompilierten Code zu springen, oder eine Verzweigungsinstruktion zu protokollieren. Diese Subroutine wird insbesondere in **Fig. 11** veranschaulicht.

Angesichts des Obigen wird, um das Verzweigungsprotokollverfahren mit dem L1-Cache auszuführen, zuerst das Verfahren in Schritt S400 gestartet. In Schritt S401 prüft der Interpretierer **110** zuerst auf einen kompilierten Codeeinsprungpunkt für das Verzweigungsziel (d. h. ob das betreffende Segment vorher kompiliert wurde). Wenn ein Einsprungpunkt vorliegt, d. h. "Ja", gibt es ein kompiliertes Segment, und der Fluß springt zu Schritt S402 zur unmittelbaren Ausführung des kompilierten Codesegments. Die Ausführung des kompilierten Codesegments geht dann weiter, bis eine Endflagge erreicht wird, und der Fluß springt anschließend für die Ausführung des nächsten Segments zurück. Selbstverständlich wird die Verzweigung im Verzweigungsprotokoll nicht aufgezeichnet, daß die Verzweigung bereits kompiliert wurde.

Wenn in Schritt S401 kein Einsprungpunkt vorliegt, d. h. "Nein", dann gibt es keinen kompilierten Code, welcher der Verzweigungsinstruktion entspricht. Der Fluß geht dann zu Schritt S404 weiter, und der Interpretierer 110 sieht im L1-Cache nach, um zu bestimmen, ob eine mögliche Übereinstimmung zwischen der Verzweigung und der Vielzahl von Verzweigungen besteht, die im L1-Cache gespeichert sind.

Schritt S404 bestimmt, ob eine Übereinstimmung zwischen der Verzweigung und der Vielzahl von Verzweigungen besteht, die im L1-Cache gespeichert sind. Der L1-Cache ist in eine Vielzahl von Sätzen geteilt, wobei jeder Satz mit einer eindeutigen Satznummer bezeichnet ist. Gemäß einer Ausführungsform der vorliegenden Erfindung enthält jeder Satz vier Verzweigungen.

Schritt S404 bestimmt zuerst eine Cache-Satznummer "S", die der aktuellen Verzweigungsadresse entspricht, wobei $S = (\text{branch_address} + \text{branch_destination}) \% \text{BRANCH_L1_SETS}$. Als nächstes wird jedes Element der `branch_L1_table[S]` sequentiell gegen die aktuelle Verzweigungsadresse und das Ziel geprüft. Wenn eine Übereinstimmung detektiert wird, d. h. "Ja", dann geht der Fluß zu Schritt S406 weiter, und die Felder "encountered_sub_count" (ein Feld, das angibt, wie viele Male die Verzweigung gefunden wurde) und "taken_sub_count" (ein Feld, das angibt, wie viele Male die Verzweigung eingeschlagen wurde) werden aktualisiert. Anschließend geht der Fluß zu Schritt S407 weiter.

In Schritt S407 wird bestimmt, ob die aktuelle Verzweigungsadresse größer als eine vorherbestimmte Schwellenanzahl gefunden wurde. Der bevorzugte Schwellenwert liegt in der Größenordnung von 1000 Bits. So wird das Feld `encountered_sub_count` mit dem Schwellenwert in Schritt S407 verglichen. Wenn der Schwellenwert überschritten wird, d. h. "Ja", dann geht der Fluß zu Schritt S410 weiter, und die im Cache gespeicherten Informationen für diese Verzweigung werden in das Verzweigungsprotokoll zurückgeschrieben. Wenn der Schwellenwert hingegen nicht überschritten wird, d. h. "Nein", dann geht der Fluß zu Schritt S412 weiter. Schritt S412 ist ein Ende der aktuellen Subroutine, die zu IC-FETCH02 springt, d. h. dem Einsprungpunkt des Interpretierers 110.

Wenn sich die korrekte Verzweigung nicht im Cache befindet, d. h. "Nein" in Schritt S404, dann geht der Fluß zu Schritt S408 weiter, und ein `BRANCH_L1_RECORD` (d. h. der Datensatz, der alle Felder, die aktualisiert werden können, enthält, wie `encountered_sub_count` und `taken_sub_count`) im oben mit "S" bezeichneten Satz wird aus dem L1-Cache entfernt und in das Verzweigungsprotokoll geschrieben. Als nächstes werden die aktuellen Verzweigungsinformationen in den mit "S" bezeichneten Satz geschrieben. Außerdem wird während des Schreibens des aktuellen Verzweigungsdatensatzes in den Satz "S" der aktuelle Verzweigungsdatensatz als erstes Element des Satzes platziert. Dies ist darauf zurückzuführen, daß dieselbe Verzweigung wahrscheinlich erneut ausgeführt wird, wodurch die Leistung und Effizienz des Systems zunehmen. Mit anderen Worten werden die Sätze S404 rascher ausgeführt. Auch wenn sich die Verzweigung im Cache befindet, d. h. "Ja", kann sie in das Verzweigungsprotokoll kopiert werden, wenn sie eine große Anzahl von Malen ausgeführt hat, seit sie das letzte Mal geräumt wurde.

Wenn der L1-Cache verwendet wird, ist die Schrittsequenz fast immer S400, S404, S406, S407 und S412. Demgemäß versucht die vorliegende Erfindung, diese Schritte so schnell wie möglich zu machen. Wenn die aktuellen Verzweigungsinformationen in das erste Element des Satzes gesetzt werden, machen die Verzweigungsinformationen den Schritt S404 schneller, da der Interpretierer 110 dieselbe Verzweigung wahrscheinlich erneut ausführt.

Das oben angegebene Verzweigungsprotokollverfahren reduziert die Belastung für den Prozessor durch das Ausführen eines Codes, der vorher kompiliert wurde, und das Erweitern eines Zugangs zu oft aufgerufenen Verzweigungsinstruktionen, welche die Schwellenhöhe zur Kompilierung noch nicht erreicht haben. In dieser Hinsicht ist der Hauptzweck des OOC, den Schritt S400 dazu zu bringen, die "Ja" Verzweigung nahezu jedesmal einzuschlagen. Wenn eine Verzweigung häufig ausgeführt wird, dann sollte ein kompiliertes Codesegment für ihr Ziel vorliegen.

Ein zweiter Zweck ist, den "Nein"-Pfad, der Schritt S401 folgt, schneller zu machen, so daß Verzweigungen, die noch nicht kompiliert wurden, die Programmausführung nicht merkbar verlangsamen. Der langsamste Teil des "Nein"-Pfades wird als "Räumung" bezeichnet. In beiden Schritten S408 und S410 werden Verzweigungsinformationen aus dem L1-Cache "geräumt" und in das Verzweigungsprotokoll geschrieben. Es wird notwendig, die Informationen einer Verzweigung zu räumen, um einen Startparameter an den Kompilierer zu senden; der veranlaßt, daß ein kompilierter Code generiert wird, und den Schritt S400 veranlaßt, für diese Verzweigung in der Zukunft "Ja" zu antworten.

Es ist jedoch nicht notwendig, die Informationen der Verzweigung jedesmal zu räumen, wenn eine nicht-kompilierte Verzweigungsadresse ausgeführt wird. Eine Räumung alle 100 Ausführungen oder weniger ist oft OK. Daher versucht die vorliegende Erfindung, die Geschwindigkeit der Schritte S400, S404, S406, S407 und S412 zu steigern, die keine Räumungen enthalten. So wird immer der schnellere Pfad eingeschlagen, außer es passiert eines von zwei Dingen. In Schritt S404 ist es möglich, daß die Verzweigungsinformationen im Satz nicht gefunden werden, daher wird der "Nein"-Pfad zu S408 eingeschlagen. Wenn die Verzweigung öfter als die "Schwellen"anzahl von Malen ausgeführt wurde, wird in Schritt S407 der "Ja"-Pfad zu S410 eingeschlagen, der auch eine Räumung enthält.

Im `OOC_DEBUG_MODE` wird das L1-Cacheverfahren weiterhin verwendet, die Schwelle zur Räumung des Cache wird jedoch auf 1 gesetzt, so daß die Informationen bei jeder Verzweigungsausführung in das Verzweigungsprotokoll geschrieben werden. Dies macht den `OOC_DEBUG_MODE` viel langsamer.

STARTPARAMETERAUSWAIL (engl. = SEED SELECTION)

Wenn eine Verzweigungsinstruktion sehr häufig ausgeführt wird, sendet der Verzweigungsprotokollierer 112 seine Zieladresse an den Kompilierer 104. Diese Adresse wird "Startparameter" genannt, und das Auswählen von Startparametern ist ein sehr wichtiger Teil des OOC-Systems.

Startparameter sollten Adressen sein, die am Beginn einer Prozedur oder am Kopf einer Schleife stehen. Daher sendet der Verzweigungsprotokollierer 112 nur Startparameter, die das Ziel einer unbedingten Verzweigung sind. Startparameter sollten Adressen sein, die häufig ausgeführt werden, so daß ein Verzweigungsziel nur ein Startparameter wird, wenn sein Feld `encountered_count` größer als eine Schwelle ist. Die Schwelle wird im OOC-Puffer im mit `seed_production_threshold` bezeichneten Feld gespeichert. Die Schwelle kann sich mit der Zeit ändern, was im nächsten Abschnitt beschrieben wird.

Es gibt zwei nachteilige Dinge bei der Verwendung einer festgelegten Schwelle, um zu entscheiden, ob ein Startparameter gesendet wird. Erstens kann die Schwelle zu hoch sein, während der Kompilierer 104 untätig ist. In diesem Fall gibt es für den Kompilierer 104 nützliche Arbeiten zu verrichten, aber der Verzweigungsprotokollierer 112 sagt dem Kompilierer 104 nicht, was zu tun ist. Zweitens kann die Schwelle zu niedrig sein, während die Nachrichtenwarteschlange voll ist. In diesem Fall versucht der Verzweigungsprotokollierer 112 einen Startparameter zu senden, obwohl der Startparameter nicht in die Warteschlange paßt, was Zeitverschwendung ist.

Glücklicherweise ist es möglich, die beiden Situationen zu detektieren, wenn der Kompilierer 104 untätig ist, und wenn die Nachrichtenwarteschlange voll ist, und die Schwelle zu ändern. Der Verzweigungsprotokollierer 112 detektiert in der Prozedur `branch_Update_Entry` durch das Lesen des als `num_monitor_seed_messages` bezeichneten OOC-Bufferfeldes, daß der Kompilierer 104 untätig ist. Wenn dieses Feld 0 ist, hat der Kompilierer 104 alle Startparameter, die gesendet wurden, beendet. Die Schwelle ist zu hoch, daher senkt sie der Verzweigungsprotokollierer 112. Der Verzweigungsprotokollierer 112 detektiert eine volle Nachrichtenwarteschlange in der Prozedur `branch_Send_Seed`, wenn er einen Startparameter zu senden versucht, und bekommt einen Fehlercode, der anzeigt, daß die Nachricht nicht gesendet wurde. Die Schwelle ist zu niedrig, daher erhöht sie der Verzweigungsprotokollierer 112.

Im `OOC_DEBUG_MODE` ändert sich die Schwelle nie. Ihr Wert wird in diesem Fall auf das dritte Argument der `OOC_INIT`-Prozedur gesetzt.

MEHRAUFGABEN-BEHANDLUNG (engl. = HANDLING MULTITASKING)

Der OOC läuft auf einem Mehrprozessorsystem mit mehrfachen Interpretierern 110. Diese Aufgaben haben individuelle Verzweigungs-L1-Caches, sie verwenden jedoch dieselbe Verzweigungsprotokollertabelle. Wenn Verzweigungsinformationen aus dem L1-Cache in die Verzweigungsprotokollertabelle geräumt werden, belegt der Interpretierer 110 ein Protokoll auf der Tabelle, so daß es zu keinem Konflikt mit irgendeiner anderen Ausführung kommt. Es gibt zwei mögliche Wege, einen Streit um die Verzweigungsprotokollverriegelung zu behandeln. Der erste ist, einen Interpretierer 110 warten zu lassen, bis die Verriegelung verfügbar ist, und dann die Verriegelung zu erhalten und ihre Verzweigungsinformationen zu schreiben. Dies läßt den Interpretierer 110 langsamer laufen, macht jedoch das Verzweigungsprotokoll genauer. Der zweite ist aufzugeben, ohne die Verzweigungsinformationen zu schreiben, wenn der Interpretierer 110 die Verriegelung nicht erhalten kann. Dieser Weg macht den Interpretierer 110 schneller, verliert jedoch einige Verzweigungsprotokollinformationen. Der OOC verwendet den zweiten Weg, da die Geschwindigkeit des Interpretierers 110 wichtiger ist als die Genauigkeit des Verzweigungsprotokolls. Die Verzweigungsprotokollinformationen müssen nur ungefähr korrekt sein, damit das System gut funktioniert.

Wenn der OOC mit mehrfachen Interpretierern 110 läuft, ist eine der Aufgaben die spezielle Masteraufgabe, die `OOC_INIT` aufruft, um den OOC-Buffer und die Verzweigungsprotokoll-Datenstrukturen zu initialisieren. Die anderen Aufgaben sind Slaveaufgaben, die nur einige lokale Variablen und ihre Verzweigungs-L1-Caches initialisieren müssen. Die Slaveaufgaben rufen `SlaveOOC_Init` auf, nachdem die Masteraufgabe die Initialisierung des `OOC-Buffer` beendet hat. Die Synchronisation zwischen Master- und Slaveaufgaben verwendet die folgenden Verfahren.

Masterverfahren

1. MCD-Instruktion ausführen, um OOC einzuschalten.
2. `OOC_INIT` aufrufen, wodurch der OOC-Buffer und Verzweigungsprotokoll-Datenstrukturen initialisiert werden.
3. Slaveaufgaben aufwecken.
4. Zum Interpretierer springen.

Slaveverfahren

1. Gehe zu Schlafmodus. Aufwachen, wenn Masteraufgabe ausführt (Schritt 3 oben).
2. `SlaveOOC_Init` aufrufen, wodurch der individuelle Verzweigungs-L1-Cache der Aufgabe initialisiert wird.
3. Zum Interpretierer springen.

BENUTZER/SYSTEMRAUMÜBERGÄNGE

Das OOC-System kompiliert nur Instruktionen von den Systemseiten des ASP-Adressenraums. Es ignoriert die Benutzerseiten. Das `OOCSTS`-Bit des individuellen Bereichs des Interpretierers 110 steuert, ob der Verzweigungsprotokollierer 112 aufgerufen wird oder nicht. Dieses Bit wird hauptsächlich von den beiden Makros `NEXT_CO` und `NEXT_OUN` gesteuert. Es gibt jedoch einen Fall, wo der OOC-Code dieses Bit setzen muß. Wenn ein kompiliertes Codesegment mit einer nicht-festgelegten Verzweigungsinstruktion endet, kann es `PSW_IA` veranlassen, sich vom Systemraum zum Benutzerraum zu bewegen, was erfordert, daß `OOCSTS` auf 0 gesetzt wird. Daher springt ein kompiliertes Codesegment, das mit einer nicht-festgelegten Verzweigung endet, zur Routine `branch_Exit_Log`, welche die Zieladresse prüft und das `OOCSTS`-Bit korrekt setzt.

ÜBERGANG ZUM/VOM KOMPILIERTEN CODE

5 Der Interpretierer 110 transferiert die Ausführung zum kompilierten Code, wenn der Interpretierer 110 eine Verzweigungsprotokollieroutine aufruft, und er ein kompiliertes Codesegment für das Verzweigungsziel findet (siehe Fig. 11). Wenn die Segmentverriegelung ausgeschaltet ist, springt der Interpretierer 110 direkt zum Einsprungpunkt. Wenn die Segmentverriegelung eingeschaltet ist, muß der Interpretierer 110 versuchen, das Segment zu verriegeln, bevor er zum Einsprungpunkt springt. Wenn er das Segment verriegelt, dann springt er zum Einsprungpunkt. Wenn er das Segment nicht verriegeln kann, springt er zurück zum Interpretierer 110.

10 Es gibt einige Wege zur Ausführung, um ein kompiliertes Codesegment zu verlassen, die in Tabelle 4 beschrieben werden. Wenn die Steuerung zum Interpretierer 110 zurückspringt, haben in allen Fällen die ESI- und EDI-Register korrekte Werte, und der individuelle Bereich des Interpretierers 110 hat perfekten K-Status.

Tabelle 4

Wie die Steuerung ein kompiliertes Codesegment verläßt

20	End-K-OP-Code	Was der kompilierte Code tut
25	Festgelegte Verzweigung oder geradliniger K-OP-Code	Testet, ob die Zieladresse einen kompilierten Einsprungpunkt hat. Wenn sie einen hat, dann wird ein Intersegmentsprung zum Einsprungpunkt durchgeführt. Wenn sie keinen hat, dann wird die Steuerung zum Interpretierer 110 bei IC_FETCH02 oder zu branch_Exit, wenn die Segmentverriegelung ein ist, zurückgeführt.
35	Nicht-festgelegte Verzweigung	Springt zu branch_Exit_Log, wodurch das OOCTSTS-Bit gesetzt und dann der Verzweigungsprotokollierer 112 aufgerufen wird, wenn PSW_IA noch in einer Systemseite ist.
45	LPSW, SSM, STNSM, MCD, CALL, RRTN, SVC, MC, BPC, LINK, LINKD, LOAD, LOADD, DELT, DELTD, FBFCC	Ohne Segmentverriegelung: springt zu IC_FETCH02, um den OP-Code auszuführen. Mit Segmentverriegelung: springt zu branch_Exit-Interpret.
50		
55		
60	SAM OP-Code, der zum RISC-Modus schaltet	Ohne Segmentverriegelung: springt zu IC_FETCH02, um den SAM OP-Code auszuführen. Mit Segmentverriegelung: springt zu branch_Exit-Interpret.
65		

Wenn die Segmentverriegelung ein ist, hält der Interpretierer 110 eine Verriegelung auf dem kompilierten Codeseg-

ment, während er diesen Code ausführt. Er muß diese Verriegelung freigeben, nachdem das Segment verläßt, so daß der kompilierte Code einige Prozeduren im Verzweigungsprotokollierer 112 aufruft, welche die Verriegelung freigeben, und dann zum Interpretierer 110 springen.

UNTERBRECHUNGEN

Es gibt einige Unterbrechungen, die eintreten können, während der kompilierte Code ausführt, wie IO-Unterbrechungen oder MCHK-Unterbrechungen. Der kompilierte Code prüft das INTINF-Feld des individuellen Bereichs, um zu detektieren, ob eine Unterbrechung eingetreten ist. Er prüft dieses Feld innerhalb irgendeiner möglichen unendlichen Schleife, die sicherstellt, daß er die Unterbrechung nicht für immer ignoriert. Wenn eine Unterbrechung eintritt, ruft der kompilierte Code die Interpretierer 110-Routine IU_OINTCIHK mit perfektem K-Status auf. Er erwartet, daß der Interpretierer 110 zum kompilierten Code zurückspringt.

INTERPRETIERERRÜCKRUF

Einige K-OP-Codes werden vom OOC nicht übersetzt.

Statt dessen ruft der kompilierte Code die Interpretierer 110-Subroutine IC_OOC auf, um den OP-Code zu interpretieren und zum kompilierten Code zurückzuspringen. Der kompilierte Code stellt sicher, daß die EST- und EDI-Register die korrekten Werte aufweisen, und daß der individuelle Bereich perfekten K-Status hat, bevor IC_OOC aufgerufen wird.

Wenn der Interpretierer 110 während der Ausführung der IC_OOC-Subroutine einen Fehler detektiert, ruft er die Prozedur OOC_EXCP auf, und springt nicht zum kompilierten Code zurück. Wenn die Segmentverriegelung eingeschaltet ist, gibt OOC_EXCP die Segmentverriegelung frei.

AUSNAHMEN

Wenn ein übersetzter OP-Code eine umaskierte Ausnahme, wie eine Operationsausnahme oder eine Nullteiler Ausnahme, aufweist, ruft der kompilierte Code eine Interpretierer-Subroutine IC_PGMxx auf, wobei xx die Fehlercodenummer zwischen 01h und 21h ist. Der Interpretierer 110 versucht, die Ausnahme zu behandeln und zurückzuspringen. Wenn der Interpretierer 110 nicht zurückspringen kann, ruft er OOC_EXCP auf, wodurch jede Segmentverriegelung freigegeben wird.

VERWENDUNG GLOBALER FUNKTIONEN

Einige K-OP-Codes, wie Zeichenverarbeitungsoperationen, werden in eine große Anzahl von Ziel-OP-Codes übersetzt. Die Herstellung mehrfacher Übersetzungen dieser OP-Codes würde zu viele der Segmentspeicher-Resubrutinen verwenden, die globale Funktionen genannt werden, und die der kompilierte Code aufruft, um diese OP-Codes auszuführen. Diese globalen Funktionen sind genauso wie Interpretierer 110-Routinen, die K-OP-Codes ausführen, außer daß sie speziell geschrieben sind, um vom kompilierten Code aufgerufen zu werden, und zum kompilierten Code zurückzuspringen. Es gibt globale Funktionen für fünf OP-Codes, SBE, CC, MV, TS und C. Versuche zeigen, daß die globalen Funktionen viel schneller sind als der Aufruf des IC_OOC-Einsprungpunkts des Interpretierers 110, und sie verwenden viel weniger Speicher als das mehrfache Kompilieren des OP-Codes in Zielinstruktionen.

VI. KOMPIlierER

ÜBERBLICK

Bevor auf die Details der Kompilierung eingegangen wird, ist es wichtig, auf hoher Ebene den Hauptzweck des Kompilierers 104 zu verstehen, und die Struktur des Kompilierers 104 zu verstehen. Der Zweck des Kompilierers 104 ist die Übersetzung häufig ausgeführter Teile des aktuellen Ausführungsprogramms in einen optimierten Zielcode, und diesen Code für den Interpretierer 110 zur Ausführung verfügbar zu machen.

Fig. 12 veranschaulicht insbesondere eine allgemeine Struktur eines Kompilierers 104. Der Kompilierer 104 empfängt Startparameter vom Verzweigungsprotokollierer 112 (oben diskutiert), die den Kompilierungsprozeß starten. Der Startparameter ist die Adresse einer Originalinstruktion, die das Ziel einer großen Anzahl von Verzweigungen im aktuellen Ausführungsprogramm gewesen ist. Dies soll einen Startpunkt bilden, um einen häufig ausgeführten Teil des aktuellen Ausführungsprogramms zu finden. Der Blockwähler 114 verwendet diesen Startparameter zusammen mit anderen Informationen, die vom Verzweigungsprotokollierer 112 vorgesehen werden, um Abschnitte des Programms zu wählen, die kompiliert werden sollten.

Sobald der zu kompilierende Originalcode gewählt wurde, wird er drei Hauptstufen unterworfen. Die erste Stufe ist die Konvertierung der K-OP-Codes in eine Zwischensprache (IL), die vom Rest des Kompilierers 104 verwendet wird. Die Zwischensprache wird vom IL-Generator 124 generiert. Die zweite Stufe führt verschiedene Analysen und optimierende Transformationen an der IL mittels der oben angegebenen Optimierung durch, die für Referenzzwecke als Optimierer 126 bezeichnet wird. Die Endstufe konvertiert die IL in einen relativierbaren Maschinencode und wird als optimierende Codegenerierungseinheit 118 bezeichnet.

Der End-Job des Kompilierers 104 ist, den optimierten Code für den Interpretierer 110 verfügbar zu machen. Dann wird eine Segmentdatenstruktur mit einer Kopie des optimierten Codes durch eine Segmentinstallationseinheit geschaffen. Anschließend wird das Segment in einen gemeinsam genutzten Bereich innerhalb des OOC-Puffers (nicht dargestellt) installiert. Die Übersetzungstabelle wird schließlich aktualisiert, so daß alle Verzweigungen vom Interpretierer 110

zum kompilierten K-Code statt dessen den neuen Zielcode verwenden.

Der Rest dieses Abschnitts diskutiert detailliert jede der obigen Kompilierer-104-Stufen. Am Ende des Abschnitts werden auch einige andere verschiedene Implementationseinzelheiten erörtert.

BLOCKWAHL (engl. = BLOCK PICKING)

Der Kompilierer 104 empfängt eine einzelne Startparameteradresse, um die Kompilierung zu starten. Beginnend beim Startparameter liest er Originalinstruktionen, bis er etwas wie einen Prozedurrumpf gelesen hat. Dann reicht er diesen Satz von Originalinstruktionen zur nächsten Kompilierer-104-Stufe, der IL-Generierung, weiter. Die Instruktionseinheiten, die der Kompilierer 104 liest, werden Basisblöcke genannt, so daß diese Stufe als Blockwähler, d. h. Blockwähler 114, bezeichnet wird.

Ein Basisblock ist eine Sequenz von Instruktionen, wo die Steuerung nur bei der ersten Instruktion einspringen kann, und nur bei der letzten Instruktion ausspringen kann. Das bedeutet, daß nur die erste Instruktion das Ziel einer Verzweigung sein kann, und nur die letzte Instruktion eine Verzweigungsinstruktion sein kann. Es bedeutet auch, daß, wenn die erste Instruktion des Blocks ausgeführt wird, dann alle Instruktionen ausgeführt werden.

BLOCKWÄHLER

Fig. 13 veranschaulicht ein Beispiel des Blockwählers 114 gemäß einer Ausführungsform der vorliegenden Erfindung. Die Prozedur OOC_ParseFrom implementiert den Blockwähler 114. Er liest einen Basisblock zu einer Zeit. Ein Basisblock endet aus einem von fünf Gründen.

1. Wenn der Sprachanalysierer eine Verzweigungsinstruktion liest, endet der Block mit der Verzweigung.
2. Wenn die nächste Instruktion bereits einer Sprachanalyse unterzogen wurde, dann endet der Block mit der aktuellen Instruktion, da jeder K-OP-Code nur einmal in einem Segment auftreten sollte.
3. Wenn die nächste Instruktion ein Verbindungspunkt ist, dann endet der Block mit der aktuellen Instruktion, da sich Verbindungspunkte am Beginn eines Basisblocks befinden müssen.
4. Wenn die aktuelle Instruktion ein Faktor ist, und darauf Daten anstelle von Instruktionen gefolgt werden könnten, dann endet der Block mit der aktuellen Instruktion.
5. Wenn die aktuelle Instruktion eine illegale Instruktion ist, dann endet der Block mit der aktuellen Instruktion.

Nach dem Lesen jedes Blocks entscheidet der Blockwähler 114 in Abhängigkeit von der Weise, wie der Block geendet hat, welche Aktion als nächstes zu unternehmen ist. Die möglichen Aktionen sind in Tabelle 5 veranschaulicht.

Tabelle 5

Aktion nach dem Lesen eines Blocks

Ende des aktuellen Blocks	Aktion des Blockwählers 114
Bedingte Verzweigung	Sprachanalyse an der Fehlschlag-Instruktion und der Verzweigungszielinstruktion fortsetzen.
Unbedingte festgelegte Verzweigung	Sprachanalyse an der Verzweigungszielinstruktion fortsetzen.
Nicht-festgelegte Verzweigung	Sprachanalyse stoppen, da Verzweigungsziel unbekannt ist.
Faktor der Endinstruktion oder illegale Instruktion	Sprachanalyse stoppen, da das nächste Byte keine Instruktion sein könnte.
Andere Instruktionen	Sprachanalyse an der Fehlschlag-Instruktion fortsetzen.

Ein Beispiel wird in Fig. 13 veranschaulicht. Der Blockwähler 114 beginnt an der Startparameterinstruktion, die eine

LB-Instruktion ist. Da dies keine Verzweigungs- oder Endfaktorinstruktion ist, geht er zur nächsten Instruktion weiter. Diese ist eine TH-Instruktion, die eine bedingte Verzweigung ist. Der Blockwähler 114 stoppt das Lesen des Blocks wegen der bedingten Verzweigung. Er setzt das Lesen neuer Blöcke sowohl an der LH- als auch der LF-Instruktion fort. Wenn er die SVC-Instruktion liest, beendet der Blockwähler 114 diesen Block, da SVC eine Endfaktorinstruktion ist. Wenn er die GO Instruktion liest, beendet der Blockwähler 114 diesen Block, da GO eine Verzweigungsinstruktion ist. Er setzt das Lesen an der L8-Instruktion fort, da sie ein Verzweigungsziel ist. Nachdem er die ST8-Instruktion liest, beendet der Blockwähler 114 den Block, da er bereits die nächste Instruktion gelesen hat.

Es gibt eine Obergrenze für die Anzahl von Instruktionen, die der Blockwähler 114 liest. Der Zweck dieser Grenze ist zu verhindern, daß dem Kompilierer 104 während der Kompilierung der Quelleninstruktionen der Speicher ausgeht. Die Grenze wird von der Konstante MAX_KINST_NUM in OOCt_trace.c gesetzt, und sie beträgt derzeit 500.

Der Blockwähler 114 kann einen Seitenfehler verursachen, wenn er versucht, eine Instruktion zu lesen. Wenn er einen Seitenfehler erhält, stoppt der Blockwähler 114 das Lesen des aktuellen Blocks, setzt jedoch das Lesen ab irgendeinem Verzweigungsziel fort, das er noch nicht versucht hat. Dies ermöglicht dem Kompilierer 104, ein Segment zu schaffen, auch wenn er nicht alle Instruktionen, die von einem Startparameter erreicht werden können, einer Sprachanalyse unterziehen kann.

BLOCKAUFBAU (BLOCK LAYOUT)

Nach der Auswahl der Basisblöcke ruft der Blockwähler die Prozedur OOCt_GenerateIL auf, um die IL-Instruktionen zu generieren, die der Rest des Kompilierers 104 verwenden wird. Zu dieser Zeit ist es möglich, die Reihenfolge der Blöcke umzuordnen. Dies wird Blockaufbau genannt, und es hilft dem Kompilierer 104, einen besseren Code für den Pentium Pro-Prozessor zu produzieren, da der Pentium Pro schneller läuft, wenn bedingte Vorwärtsverzweigungen nicht eingeschlagen werden.

Das Beispiel in Fig. 13 wird herangezogen. Es hat eine bedingte Verzweigung, die TH-Instruktion. In den Originalinstruktionen ist der Fehlschlag-Basisblock derjenige, der mit LH beginnt, und der Zielblock ist derjenige, der mit LF beginnt. Wenn die bedingte Verzweigung 75% der Zeit eingeschlagen wird, dann läuft er schneller, wenn der LF-Basisblock in die Fehlschlag-Position und der LH-Basisblock in die eingeschlagene Verzweigungsposition gebracht wird.

Die OOCt_GenerateIL-Prozedur baut die Blöcke gemäß den Informationen im Verzweigungsprotokoll auf. Sie bringt die üblichsten Nachfolger bedingter Verzweigungen in die Fehlschlag-Position, wann immer sie kann. Diese Prozedur produziert eine Liste von IL-Instruktionen, die zu den Optimierungsphasen des Kompilierers 104 weitergereicht werden.

GENERIERUNG DER ZWISCHENSPRACHE (IL)

Der Abschnitt diskutiert den Prozeß der Generierung der Kompilierer-104-Zwischensprache(IL)-Repräsentation für die K-Codes. Vor der direkten Erörterung, wie die IL generiert wird, wird ein Überblick über die IL gegeben, und Datenstrukturen, die wichtig zu verstehen sind, werden beschrieben.

IL-ÜBERBLICK

Die Hauptanalyse- und Transformationsdurchläufe des Kompilierers 104 operieren in einer Zwischensprache, die ein spezieller maschinenunabhängiger Instruktionssatz ist. Die Verwendung einer Zwischensprache ist aus zwei Hauptgründen eine Standard-Kompilierer-104-Technik. Erstens hat eine IL typischerweise eine Architektur, die eine Analyse und Transformationen vereinfacht. Zweitens ermöglicht eine IL vielen verschiedenen Quellsprachen, dieselben Optimierungs- und Codegenerierungsstufen zu verwenden, und erleichtert die erneute Zielrichtung auf verschiedene Plattformen.

Die vom OOCt verwendete IL (ab hier nur als IL bezeichnet) ist derzeit aus 40 verschiedenen OP-Codes zusammengesetzt, die in Tabelle 6 aufgelistet sind. Die Instruktionen fallen in drei Hauptkategorien. Erstens gibt es funktionelle OP-Codes, wie ADD und LOAD, die ein geradliniges Mapping auf Standard-Maschinen-OP-Codes aufweisen. Zweitens gibt es OP-Codes, die den Steuerfluß behandeln, wie LABEL und CGOTO. Schließlich gibt es eine Anzahl spezieller OP-Codes, welche als spezielle Marker vom Kompilierer 104 verwendet werden, die nicht direkt dem Code entsprechen, der vom hinteren Ende generiert wird. Diese speziellen Marker-OP-Codes werden in einem getrennten Abschnitt beschrieben. Da die IL eine virtuelle Maschine repräsentiert, ist es geradlinig, andere OP-Codes zur IL hinzuzufügen, wenn eine weitere Funktionalität erforderlich ist.

Die IL besteht aus Instruktionen, von denen jede einen der OP-Codes, einen Typ und eine Anzahl von Pseudoregisterargumenten spezifiziert. Die vom Kompilierer 104 unterstützten Typen sind 8 Bit-, 16 Bit- und 32-Bitwerte mit und ohne Vorzeichen. Abgesehen von Zwischenwerten, die vom SET-OP-Code verwendet werden, und Werten, die mit dem LOAD-OP-Code aus dem Speicher geladen werden, werden alle Argumente mit Pseudoregistern weitergereicht. Pseudoregister sind einfach die Register der virtuellen IL-Maschine. Der Kompilierer 104 gestattet eine willkürliche Anzahl von Pseudoregistern, von denen jedes eine vordefinierte Größe (z. B. 16 Bits) hat. Jedes Pseudoregister entspricht direkt einer spezifizierten Speicherstelle. Für den OOCt sind diese Speicherstellen in den OOCt-spezifischen Teilen des individuellen Bereichs. Dieses Mapping von Pseudoregistern in Speicherstellen hat zwei Vorteile. Erstens wird die IL rationalisiert. Die IL-Operationen, um allgemein verwendete Werte in temporäre Speicher zu laden, und sie in den Speicher zurückzuladen, sind nicht erforderlich. Zweitens kann der Kompilierer 104 oft allgemein verwendete Werte Maschinenregistern zuordnen, wodurch redundantes Laden oder Speichern eliminiert wird.

Tabelle 6

IL-OP-Codes

5	OP-CODE	BESCHREIBUNG
	LABEL	Markiert einen Platz im Flußgraphen, der das Ziel von Sprungoperationen sein könnte
	GOTO	Ein Sprung zu einer Sprungmarke bzw. engl. label
	C'GOTO	Ein bedingter Sprung zu einer Sprungmarke auf der Basis des Booleschen Werts eines Pseudoregisters
10	IGOTO	Ein indirekter Sprung zu einer Adresse, die durch den Wert eines Pseudoregisters bestimmt wird
15	OP-CODE	BESCHREIBUNG
	SET	Setzt einen unmittelbaren Wert in ein Pseudoregister
	ASSIGN	Bewegt den Wert in einem Pseudoregister in ein anderes Pseudoregister
	OASSIGN	Spezielle Markerinstruktion, die zeigt, wo Pseudoregister überlappen, um ein Aliasing explizit zu machen
20	C'VT	Ein Pseudoregister von einem Typ in einen anderen konvertieren (z. B. Zeichenerweiterung, Nullerweiterung)
	NEG, CMPL, BSWAP	Unäre Negation, logisches Komplement, Byte-Swap
	ADD, SUB, MUL, DIV,	Binäres Addieren, Subtrahieren, Multiplikation, Dividieren, Rest
25	REM	
	ASL, ASR	Arithmetische Verschiebung links, rechts
	LSR	Logische Verschiebung rechts
	BAND, BOR, BXOR	Binäres logisches Und, Oder, Xoder
	EQ, NE, LT, LE, GT, GE	Vergleicht zwei Eingabeoperanden und setzt Ausgangsoperanden auf wahr, wenn op1 == op2, op1 != op2, op1 < op2, op1 <= op2, op1 > op2, op1 >= op2
30	TESTZ, TESTNZ	Vergleicht zwei Eingabeoperanden und setzt Ausgangsoperanden auf wahr, wenn (op1 & op2) == 0, (op1 & op2) != 0
	C'MP	Vergleicht zwei Eingabeoperanden und setzt Ausgangsoperanden auf -1, wenn op1 < op2, auf 0, wenn op1 == op2, und auf 1, wenn op1 > op2. Dies wird aktuell vom OOC nicht verwendet.
35		
40	OP-CODE	BESCHREIBUNG
	LOAD	Laden eines Pseudoregisters mit einem Wert aus einer spezifizierten Speicherstelle
	STORE	Speichern des Werts eines Pseudoregisters in eine spezifizierte Speicherstelle
	GCALL	Führt einen Funktionsaufruf an eine von einem Satz vorherbestimmter globaler Funktionen durch
45	ICALL	Führt einen indirekten Funktionsaufruf durch, ähnlich IGOTO
	EXT	Ausgang aus dem kompilierten Block. Dies wird derzeit vom OOC nicht verwendet.
	ENTRY	Markiert einen Punkt, wo die Steuerung in den Flußgraphen einspringen kann.
	SYNC	Markiert die Punkte, wo ein Satz von Pseudoregistern in den Speicher geräumt wird
	EXTMOD	Markiert ein Pseudoregister als extern modifiziert. Dies wird verwendet, um eine Modifikation von Pseudoregistern durch Funktionsaufrufe zu behandeln.
50	SETCC	Setzt einen Booleschen auf den Wert eines Bedingungscode auf der Basis einer Operation. Dies wird eingesetzt, um Plätze zu repräsentieren, wo Flaggen verwendet werden. Derzeit werden alle SETCC-Operationen in den Nachfolger gefaltet, so daß sie nicht emittiert werden, aber die Verwendung von SETCC macht den Fluß des Werts des Bedingungscode expliziert, ohne daß der Kompilierer 104 mehrfache Ziele für eine einzelne IL-Operation repräsentieren muß.
55		

SPEZIELLE IL-OP-CODES

60 Die OOC-IL enthält einige OP-Codes, die spezielle Zwecke haben. Die meisten IL-OP-Codes entsprechen dem Code, der im hinteren Ende generiert wird. Statt dessen arbeiten diese speziellen Instruktionen als Wegweiser für den Kompilierer 104, daß etwas Spezielles geschieht. Die IL enthält die folgenden speziellen OP-Codes: ENTRY, SYNC, EXTMOD und OASSIGN. Dieser Abschnitt diskutiert die ersten drei dieser OP-Codes. OASSIGNs werden nachstehend

65 umfassend erläutert.
Der OP-Code ENTRY markiert einen Punkt, wo die Steuerung in den Flußgraphen einspringen kann. Der vom OOC generierte Code kann mehrfache externe Einsprungpunkte aufweisen, die externe Verbindungspunkte repräsentieren. Jeder der externen Einsprünge hat eine entsprechende ENTRY IL-Instruktion. Die ENTRY-Instruktionen treten am Ende

des Codes auf und werden unmittelbar von einer GOTO-Instruktion gefolgt, die zu einer Sprungmarke innerhalb des Haupttruppfes des Codes springt. Der Grund, daß ein Einsprung verwendet wird, anstatt daß der externe Einsprung direkt zur Sprungmarke springen gelassen wird, ist, es dem Codegenerator zu ermöglichen, Füllzeichen zwischen ENTRY und dem Sprung zur Sprungmarke einzufügen.

Fig. 14 veranschaulicht eine Übersicht über einen Code mit zwei externen Einsprungpunkten, wo Füllzeichen zwischen der ENTRY-Instruktion und der GOTO-Instruktion eingefügt wurden. Mit anderen Worten veranschaulicht Fig. 14 insbesondere ein Einsprungbeispiel gemäß einer Ausführungsform der vorliegenden Erfindung.

Der OP-Code SYNC wird verwendet, um zu garantieren, daß ein Bereich von Pseudoregistern in den Speicher geräumt wird. Insbesondere verwendet der OOC' den OP-Code SYNC, um zu garantieren, daß alle K-Register in den Speicherstellen sind, wo der Interpretierer 110 sie zu finden erwartet. Der SYNC arbeitet als Direktive für den Registerzuordner, wobei sie anzeigt, daß ein Pseudoregister, das sich in einem Maschinenregister befindet, welches modifiziert wurde, geleert werden sollte. Ein SYNC arbeitet auch als Verwendung aller lebendigen Daten, was verhindert, daß der Kompilierer 104 einen toten Code eliminiert, der nur den Effekt hat, K-Register zu modifizieren.

Der OP-Code EXTMOD wird verwendet, um anzuzeigen, daß ein Pseudoregister modifiziert ist, der Kompilierer 104 verfügt jedoch über keine Einzelheiten darüber, wie das Register modifiziert wurde. So hat der EXTMOD zwei Effekte. Erstens arbeitet er als Barriere gegen Optimierungen wie Constant Folding oder Copy Propagation. Zweitens zwingt er den Registerzuordner des Kompilierers 104 dazu, Füllzeichen vor der nächsten Verwendung des Pseudoregisters einzufügen. Im OOC'T werden EXTMOD-Instruktionen nach einem Rückruf zum Interpretierer 110 verwendet, um anzuzeigen, welche K-Register modifiziert worden sein können.

IL-DATENSTRUKTUREN

Vor der Diskussion, wie die IL aus den K-OP-Codes aufgebaut wird, ist es nützlich, die im Kompilierer 104 verwendeten Hauptdatenstrukturen zu kennen.

ZONE (compiler/zone.[h,c])

Die Speicherzuordnung im Kompilierer 104 wird mit einer ZONE genannten Abstraktion behandelt. Die ZONE-Abstraktion ist ein effizienter Weg zur Speicherzuordnung, so daß er sofort freigegeben werden kann. Mit der ZONE-Abstraktion ist die Zuordnung schnell, und der Programmierer muß sich keine Sorgen über Speicherlecks machen, da eine Zerstörung der ZONE den gesamten verwendeten Speicher erneut beansprucht.

Im Kompilierer 104 wird eine ZONE geschaffen, und alle Aufrufe, die Speicher zuordnen (d. h. was normalerweise malloc-Aufrufe wären), rufen ZONE_Alloc mit der anfänglich geschaffenen ZONE auf. Wenn der Kompilierer 104 fertig ist, ruft er ZONE_Destroy auf, was die Zuordnung der gesamten ZONE rückgängig macht (d. h. äquivalent zu einem Freimachen für den gesamten Speicher).

Die zugrundeliegende Implementation der ZONE verwendet 'Stücke' des Speichers. Wenn die ZONE geschaffen wird, kann sie an einem Block mit einer Größe von 0×2000 Bytes 'malloc' ausführen. Aufrufe von ZONE_Alloc verwenden dieses 'Stück' des Speichers, bis es verbraucht ist. Wenn kein Platz ist, eine ZONE_Alloc-Anforderung mit den anfänglichen 0×2000 Bytes zu bedienen, wird ein neues 'Stück' geschaffen. Weitere ZONE_Alloc-Aufrufe verwenden dieses 'Stück', bis es auch verbraucht ist.

Im Kompilierer 104 sind die Dinge durch die Tatsache, daß der gesamte Speicher vor-zugeordnet ist, etwas kompliziert, und so kann malloc nicht aufgerufen werden. Statt dessen wird eine spezielle ZONE-Zuordnereinheit (die ZALLOC-Einheit) verwendet. Der ZONE-Zuordner wird mit einem großen Speicherpool initialisiert (beispielsweise $0 \times 10\,000$ Bytes). Er teilt den Speicher in Stücke mit gleicher Größe, welche die ZONE zur Zuordnung verwendet, und behält eine Liste freier Stücke. So werden die 'malloc'-Anforderungen durch einen Aufruf von ZALLOC_get_chunk ersetzt, der ein freies 'Stück' Speicher zurückgibt. Ähnlich werden die Aufrufe zur 'Freigabe' in ZONE_Destroy durch Aufrufe von ZALLOC_free_chunk ersetzt. In der aktuellen Implementation ist die maximale Zuordnungsgröße, die von ZONE_Alloc behandelt werden kann, die anfängliche Stückgröße. Diese Grenze könnte durch die Änderung der ZALLOC-Einheit 'festgelegt' werden, um Zuordnungen variabler Größe zu behandeln, anstatt einfach eine Größe zu behandeln (siehe Segmentzuordnungseinheit für ein Beispiel dieses Typs eines Zuordners). Es gibt zwei Gründe, warum dies hier nicht durchgeführt wurde. Erstens ist ein Zuordner mit variabler Größe viel komplexer und schafft Probleme wie eine Fragmentierung. Zweitens kann die Stückgröße mit wenig bis gar keinen Einbußen sehr groß gemacht werden. Wenn das Stück ausreichend groß ist, fordert der Kompilierer 104 nur eine einzelne Zuordnung an, die größer ist als die Stückgröße, wenn der Kompilierer 104 sowieso keinen Speicher mehr gehabt hätte. So gibt es keinen echten Vorteil bei der Generalisierung der ZALLOC-Einheit, um Zuordnungen mit variabler Größe zu behandeln.

IL_CTXT (compiler/oc_common/include/il_internal.h)

Der Kompilierer 104 hält eine einzelne Datenstruktur, die IL_CTXT, um den aktuellen Zustand der Kompilierung zu verfolgen. Die IL_CTXT-Datenstruktur speichert einen Zeiger zu einer verketteten Liste von IL_NODES, die den aktuell kompilierten Code repräsentieren. Die IL_CTXT speichert auch eine Anzahl verschiedener Felder, die während des gesamten Kompilierungsprozesses verwendet werden, wie die ZONE- und IL_FRAME-Struktur, die verwendet werden. Jede der Stufen des Kompilierers 104 hat die IL_CTXT als Argument und nimmt an dieser Datenstruktur Modifikationen vor, beispielsweise eine Anzahl der Stufen addieren, oder IL_NODES entfernen.

IL_NODE (compiler/oc_common/include/il_internal.h)

Die IL_NODE-Datenstruktur repräsentiert eine einzelne abstrakte Instruktion in der Zwischensprache des Kompilie-

ers 104, wie aus einem K-OP-Code übersetzt.

Die IL_NODES, die aus den K-OP-Codes generiert werden, werden in einer doppelt verketteten Liste gehalten. Zeiger auf die ersten und letzten Elemente in dieser Liste werden in der IL_CTXT gehalten. Diese Liste repräsentiert den Code, an dem der Kompilierer 104 aktuell arbeitet. Jeder Durchlauf des Kompilierers 104 quert diese Liste und generiert entweder Informationen über den Code in der Liste oder transformiert die Liste.

Jede IL_NODE enthält ein Operationsfeld 'op', das die Grundbeschaffenheit der Instruktion anzeigt. Sie enthält auch einen Vektor von Operandenfeldern, welche die Operanden der Instruktion repräsentieren. Die Interpretation der Operandenfelder ist vom Operationstyp der Instruktion abhängig. Zusätzlich zu den Operations- und Operandenfeldern enthalten alle IL_NODES eine Anzahl von Feldern, die von allen Knotentypen gemeinsam genutzt werden, wie K pc der Instruktion, aus welcher der Knoten übersetzt wurde, die Startadresse des für den Knoten generierten Zielmaschinencodes, etc.

Die Anzahl von Operandenfeldern in einem Knoten variiert gemäß dem Operationstyp. Tatsächlich können in einigen Fällen zwei Knoten desselben Typs eine verschiedene Anzahl von Operanden aufweisen; die Anzahl von Operanden für eine Aufrufoperation ist beispielsweise von der Anzahl von Argumenten abhängig, die zum Zielverfahren weitergereicht wurden. Diese Variation der Anzahl von Operanden bedeutet, daß IL_NODES keine konsistente Größe aufweisen, und daß der Operandenvektor das letzte Element in der IL_NODE-Struktur ist. Der Operandenvektor wird als einen Eintrag lang deklariert, und IL_NODES werden durch die Berechnung/Zuordnung des gesamten Speicherbetrags, der für die gemeinsamen Felder und die Operandenfelder notwendig ist, und durch das Umformen des zugeordneten Speichers zu einem IL_NODE-Zeiger zugeordnet.

In den meisten, jedoch nicht in allen Fällen erfordert jeder Operand tatsächlich zwei aufeinanderfolgende Einträge im Operandenvektor. Der Eingangsoperand [i] des Pseudoregisters, in dem der Operand zu finden ist. Wenn der Operand ein Zieloperand ist, zeigt Operand[i+1] zu einer Liste von Knoten, die den Wert verwenden, der von dieser Operation definiert wird; wenn der Operand ein Quellenoperand ist, zeigt Operand [i+1] zu einer Liste von Knoten, die Definitionen für den Wert enthalten.

Wenn eine Operation einen Zieloperanden aufweist, wird dieser Operand immer in Operand[0] und Operand[1] gespeichert.

Wenn Operand[i] ein Quellen-(Eingangs- oder verwendungs-)operand ist, ist dies auch Operand[i+2]; d. h. alle Quellenregister müssen an das Ende der Liste der Operanden kommen.

Auf Operandenfelder in einem Knoten wird niemals direkt zugegriffen. Der Zugriff erfolgt eher durch einen großen Satz von Makros in Form von ILOP_xxx(N), wobei N ein Zeiger zu einer IL_NODE ist. Diese Makros wissen, wie verschiedene Operanden im Operandenvektor für alle verschiedenen Instruktionstypen gespeichert werden.

Einige der Knotentypen sind wie folgt (diese Liste ist nicht vollständig):

Unäre Operationen

Diese repräsentieren eine Varietät einfacher unärer (1 Quellenoperand) Instruktionen, die eine Zuweisung enthalten.

Typ

der Typ der Operation

ILOP_DEST(N)

Zielregister, wo das Resultat hingeht

ILOP_DEST_use(N)

Liste von Instruktionen, die das Zielregister verwenden

ILOP_SRC(N)

Quellenregister

ILOP_SRC_def(N)

Liste von Instruktionen, welche die Quelle definieren

Binäre Operationen

Eine große Anzahl binärer (2 Quellenoperanden) Instruktionen werden von dieser Kategorie repräsentiert.

Typ

der Typ der Operation

ILOP_DEST(N)

Zielregister, wo das Resultat hingeht

ILOP_DEST_use(N)

Liste von Instruktionen, die das Zielregister verwenden

ILOP_SRC1(N)

erstes Quellenregister

ILOP_SRC1_def(N)

Liste von Instruktionen, welche die erste Quelle definieren

ILOP_SRC2(N)

zweites Quellenregister

ILOP_SRC2_def(N)

Liste von Instruktionen, welche die zweite Quelle definieren

ILOP_DIVEX(N)

Dieser Operand tritt nur für die DIV- und REM-Operationen auf, und zeigt auf eine (Singleton) Liste, die den Knoten enthält, der den Start der Nullteilungsausnahme repräsentiert, sofern eine vorliegt.

Eine LABEL-Instruktion repräsentiert einen Punkt im Code, wohin sich Verzweigungen verzweigen können. Sie enthält die folgenden Operanden:

ILOP_LABEL(N)

Eindeutige ganze Zahl, welche die Sprungmarke (engl. label) identifiziert.

ILOP_LABEL_refs(N)

Liste von Instruktionen, die auf diese Sprungmarke bezugnehmen

ILOP_LABEL_live(N)

BITSET, der zeigt, welche Register an dieser Sprungmarke lebendig sind.

ILOP_LABEL_rd(N)

Vektor von Listen der Definitionen jedes Registers, der diese Sprungmarke erreicht.

ILOP_LABEL_misc(N)

Platz für jeden Durchlauf, um private Informationen über die Sprungmarke anzuhängen.

GOTO

Eine GOTO-Instruktion repräsentiert eine unbedingte Verzweigung zu einer Sprungmarke.

ILOP_LABEL(N)

Eindeutige ganze Zahl, welche die Zielsprungmarke identifiziert.

ILOP_LABEL_refs(N)

Singleton-Liste der Ziel-LABEL-Instruktion.

CGOTO

Eine CGOTO-Instruktion repräsentiert eine bedingte Verzweigung zu einer Sprungmarke. Sie enthält dieselben Operanden wie die GOTO-Instruktion sowie einige zusätzliche Operanden.

ILOP_COND(N)

Register, das die Bedingung enthält, bei der zu verzweigen ist. Dieses Register muß einen Wert vom Booleschen (B1) Typ enthalten. Die Verzweigung wird eingeschlagen, wenn die Bedingung TRUE ist.

ILOP_COND_def(N)

Liste von Instruktionen, die dieses Register definieren.

ILOP_COND_live(N)

BITSET, der zeigt, welche Register lebendig sind, wenn die Verzweigung nicht eingeschlagen wird.

Zusätzlich zu den instruktionsspezifischen ILOP-Makros gibt es eine Anzahl generischer Makros, die bei jeder Instruktion verwendet werden können.

ILOP_HasDEST

Gibt TRUE (WAHRT) zurück, wenn die Instruktion ein Zielregister aufweist. In diesem Fall können die ILOP_DEST und ILOP_DEST_use-Makros bei dieser Instruktion verwendet werden.

IL_OP_START, IL_OP_DONE, IL_OPNEXT

Werden zur Iteration durch die Quellenregister einer Instruktion verwendet. IL_OP_START führt einen IL_OP_INDEX zurück, der auf das erste derartige Quellenregister bezugnimmt. IL_OP_DONE testet einen IL_OP_INDEX, um zu sehen, ob er auf ein Quellenregister bezugnimmt; true wird zurückgeführt, wenn dies nicht der Fall ist. IL_OP_NEXT wird verwendet, um zum nächsten Quellenregister weiter zu gehen.

IL_OP, IL_OP_def

Diese führen das bestimmte Quellenregister und die Definitionsliste dafür für einen gegebenen IL_OP_INDEX zurück. Diese 5 Makros werden allgemein in einer Schleife der Form verwendet:

for (op=IL_OP_START(n); !IL_OP_DONE(n,op); op=IL_OP_NEXT(n,op)) {use IL_OP(n, IL_FRAME (compiler/oc_common/include/il_frame.h, compiler/OOCT_Frame.c).

Die IL_FRAME-Datenstruktur wird verwendet, um Informationen über den Kontext zu geben, in dem der kompilierte Code laufen wird. Der Rahmen bzw. engl. frame definiert die Größe und Speicherstelle für jedes der Pseudoregister, wie die Pseudoregister andere Pseudoregister überlappen, und welche Maschinenregister im Registerzuordner legal zu verwenden sind. Zusätzlich definiert die IL_FRAME-Struktur, ob ein C-Stapelrahmen für den Code, der kompiliert wird, erforderlich ist oder nicht. Im OOCT werden keine C-Stapelrahmen verwendet.

Im Kompilierer 104 wird die IL_FRAME-Struktur durch die Funktionen in OOCT_Frame.c aktualisiert. Diese Funktionen erstellen jedes der Pseudoregister, die den K-Registern und PSW-Stellen entsprechen. Ferner werden die temporären Pseudoregister des Kompilierers 104 so gesetzt, daß sie dem Arbeitsraumbereich des Interpretierers 110 entsprechen. Es werden auch Informationen darüber erstellt, wie die K-Register einander überlappen.

NL_LIST (compiler/oc_common/[include, src]/nl_nodelist.h)

An vielen Plätzen verwendet der Kompilierer 104 Listen von IL_NODES, die NL_LIST-Datenstruktur sieht eine Abstraktion für die Manipulation dieser Knotenlisten vor. Beispielsweise schafft die oben angegebene UseDef-Analyse Listen von IL_NODES, die eine gegebene Definition verwenden, und Listen von IL_NODES, welche die Definition für eine gegebene Verwendung sein können. Die NL_LIST-Abstraktion ist geradlinig, sie sieht die Fähigkeit zum Schaffen, Entfernen, Ersetzen, Suchen und Iterieren von Knotenlisten vor.

Nachdem der oben angegebene Blockwähler 114 gewählt hat, welche K-OP-Codes zu kompilieren sind, involviert die Übersetzung der K-OP-Codes in die IL drei Hauptschritte. Der erste Schritt ist die Bestimmung der Reihenfolge, in welcher der Code für die Basisblöcke generiert wird. Das Blockaufbauverfahren ist vorstehend angegeben. Zweitens, während die Basisblöcke der K-OP-Codes durch das Blockaufbauverfahren gewählt werden, werden die OP-Codes untersucht, um zu bestimmen, ob sie in einen 'logischen OP-Code' kombiniert werden können. Schließlich wird eine IL-Generierungsprozedur auf der Basis des K-OP-Codes und seiner Argumente aufgerufen.

Opcode Combination (compiler/ooct_opcode_combine.c)

Einige Sequenzen von K-OP-Codes können als einzelner 'logischer' OP-Code beschrieben werden. Beispielsweise wurde bestimmt, daß eine Sequenz von zwei TR-Instruktionen zum Testen des Werts eines 32 Bit-Registerpaares durch das Testen jeder der einzelnen Hälften verwendet wurde. Diese beiden TR-Instruktionen repräsentieren einen logischen 32 Bit-Test-OP-Code, der in der K-Architektur nicht verfügbar ist. Der Code, den die IL-AufbauprozEDUREN für die beiden TR-Instruktionen schaffen würden, ist viel weniger effizient als der Code, der geschaffen werden könnte, wenn dieses Muster erkannt werden würde. Da der OOCT-Software ist, ist es glücklicherweise einfach, einen neuen OP-Code hinzuzufügen, eine spezielle Einheit, welche die Muster erkennt, zu haben und statt dessen die effiziente IL zu generieren.

Vor der Generierung der Standard-IL für einen gegebenen OP-Code wird die Routine OOCT_opcode_combine aufgerufen. Diese Routine wiederholt alle Muster, die definiert wurden, wobei sie versucht, einen 'logischen' OP-Code zu verwenden, wenn er geeignet ist. Derzeit sind nur zwei Muster definiert, es ist jedoch geradlinig, zusätzliche Kombinationen zu definieren. Wenn eines der Muster übereinstimmt, wird die IL-AufbauprozEDURE für diesen logischen OP-Code verwendet, um die IL-Instruktionen zu schaffen, und OOCT_opcode_combine führt TRUE zurück, um anzuzeigen, daß die normale IL-AufbauprozEDURE nicht aufgerufen werden muß.

IL_Building Procedures (compiler/ooct_il_build.c)

Für jeden K-OP-Code gibt es eine spezifische IL-AufbauprozEDURE. Die IL-AufbauprozEDUREN verwenden zwei Typen von Argumenten, die Adresse der Instruktion, und eine Liste von Argumenten, die sich in den Feldern in der Originalinstruktion befinden. Die IL-AufbauprozEDUREN verwenden auch einen gemeinsam genutzten, globalen variablen global_gen_state, der verwendet wird, um die Pseudoregister und Sprungmarken während der Generierung der IL zu verfolgen. Jede der IL-AufbauprozEDUREN fügt IL-Instruktionen zur IL_CTXT-Struktur hinzu. Alle der IL-Generierungsroutinen schaffen einen LABEL_IL_NODE mit der Adresse der Originalinstruktion als Identifikator der Sprungmarke (wenn die Sprungmarke nicht das Ziel einer anderen Instruktion ist, wird sie früh im Optimierungsprozeß eliminiert), es wird jedoch nicht allgemein versucht, Optimierungen durchzuführen, was späteren Kompilierer-104-Stufen überlassen wird, es werden aber einige Spezialfälle behandelt, wie das Prüfen auf Ausnahmen, die zur Kompilierungszeit detektiert werden können.

Die meisten der IL-AufbauprozEDUREN sind geradlinig oder direkt, sobald die IL und die Originalinstruktion des Codes generiert werden, um bekannt oder vertraut zu werden. Es gibt einige Tips, die helfen, den Code zu verstehen:

Der IL-Aufbau wurde so ausgebildet, daß die Kompilierung irgendeines gegebenen OP-Codes leicht für eine Diagnose abgeschaltet werden kann. Dies wird hauptsächlich mit dem Makro REALLY_COMPILE und den Makros COMPILE_SECTION_XX gesteuert. Wenn REALLY_COMPILE ausgeschaltet ist, bauen alle IL-AufbauprozEDUREN einfach Aufrufe (oder Sprünge) zurück zum Interpretierer 110 auf. Wenn COMPILE_SECTION_X ausgeschaltet ist, bauen alle IL-AufbauprozEDUREN für OP-Codes im Abschnitt Nummer X einfach Aufrufe (oder Sprünge) zurück zum Interpretierer 110 auf.

Da die IL typisiert ist, ist es kritisch, Pseudoregister mit der korrekten Größe mit dem korrekten Typ zu verwenden. Um beispielsweise einen 16 Bit-Wert in ein 32 Bit-Register zu laden, wird zuerst eine 16 Bit-Ladung in ein 16 Bit-Pseudoregister vorgenommen, und dann wird eine CVT-Operation verwendet, um den 16 Bit-Wert in einen 32 Bit-Wert umzuformen (das Makro LOAD_CVT32 tut dies).

Wann immer ein Rückruf oder Sprung zum Interpretierer 110 eingefügt wird, muß SYNC hinzugefügt werden, um sicherzustellen, daß der Interpretierer 110 die korrekten Werte für die K-Register aufweist. Der kompilierte Code versucht nicht, den Wert des ESI-Registers zu halten, wie es geht (tatsächlich wird er verwendet, um andere Werte zu halten). So muß der generierte Code vor einem Aufruf oder Sprung zum Interpretierer 110 den korrekten Wert in das ESI-Register setzen. Wenn ein Rückruf erfolgt, muß der Code auch eine EXTMOD-Instruktion für jedes Pseudoregister enthalten, das vom Rückruf modifiziert worden sein kann (das Makro MODIFIES_REG tut dies).

Ein Code zur Behandlung von Ausnahmebedingungen (wie Überlauf) ist nicht ausgerichtet. Statt dessen wird der Code am Ende der Liste von IL-Instruktionen generiert. Dadurch kann der allgemeine Fall als Fehlschlag kompiliert werden, was typischerweise die Leistung des generierten Codes verbessert.

Einsprungpunkte, Unterbrechungsprüfungen

Zusätzlich zur IL, die für jeden vom Blockwähler 114 gewählten K-OP-Code generiert wird, wird eine IL für Einsprungpunkte, Unterbrechungsprüfungen generiert.

Um zu ermöglichen, daß mehr Optimierungen eintreten, wird jedes Verzweigungsziel nicht als externer Einsprungpunkt eingeschlossen (externe Einsprungpunkte arbeiten als Barriere für Optimierungen). Insbesondere sind die einzigen Ziele, die zu externen Einsprungpunkten gemacht werden sollten, jene, zu denen von außerhalb des Segments gesprungen wird. Wenn ein gegebenes Segment kompiliert wird, sind Teillinformationen darüber im Verzweigungsprotokoll verfügbar, welche Ziele dieses Kriterium erfüllen (für Informationen über das Verzweigungsprotokoll siehe oben). Der

Kompilierer 104 verwendet diese Informationen, um zu wählen, welche Basisblöcke externe Einsprünge haben sollten. Für jeden dieser Einsprünge wird eine ENTRY IL_NODE zusammen mit einer GOTO IL_NODE generiert, der zur generierten IL für die Originaleinsprunginstruktion springt.

Die OOC-T-Spezifikationen zeigen an, daß der Kompilierer 104 Unterbrechungsprüfungen innerhalb irgendeiner Schleife einfügen sollte. Wenn die IL generiert wird, wird eine konservative Schätzung gemacht, indem Unterbrechungsprüfungen innerhalb jeder Rückwärts-Verzweigung innerhalb des Segments und vor irgendeiner berechneten Sprunginstruktion eingefügt werden. Die Unterbrechungsprüfung wird nach der Sprungmarke für die letzte Originalinstruktion im Basisblock eingefügt. Wie bei anderen Ausnahmebedingungen wird der IL-Code für die Unterbrechung nicht ausgerichtet generiert, so daß der Normalfall einfach der Fehlschlag der bedingten Verzweigung ist.

BESCHREIBUNG DES MITTLEREN ENDES DES KOMPILIERERS

ÜBERBLICK ÜBER DAS MITTLERE ENDE

Der Hauptzweck des 'mittleren Endes' des Kompilierers 104 ist die Verbesserung der Qualität der IL, so daß ein besserer Code in der Codegenerierungsstufe generiert wird. Der Rest des Kompilierers 104 ist als Serie von Durchläufen strukturiert, die entweder eine Analyse der IL durchführen, oder eine Transformation durchführen, welche die IL modifiziert. Die Durchläufe können mehrfache Male angelegt werden, obwohl es einige Abhängigkeiten zwischen Durchläufen gibt. Ab diesem Punkt hat der Rest des Kompilierers 104 keine besondere Kenntnis von K-Instruktionen, er hat nur mit der IL zu tun.

Der Rest dieses Abschnitts ist wie folgt unterteilt. Als erstes wird die Stufe diskutiert, welche die OASSIGN-Einfügung vornimmt. Als zweites werden Analysedurchläufe des Kompilierers 104 diskutiert. Schließlich werden Transformationsdurchläufe (welche die Hauptoptimierungen vornehmen) des Kompilierers 104 diskutiert.

Fig. 15 veranschaulicht teilweise ein OASSIGN-Einfügebungsbeispiel.

OASSIGN INSERTION (compiler/ooct_add_overlap_defs.c)

Der OASSIGN-OP-Code ist eine spezielle Markerinstruktion, die ein Aliasing zwischen Pseudoregistern explizit macht. Die Notwendigkeit von OASSIGN entsteht im OOC, da einige K-OP-Codes 16 Bit-Register verwenden, während andere Operationen 32 Bit-Register verwenden, welche als Alias der 16 Bit-Register fungieren. Im OOC werden getrennte Pseudoregister für alle der 16 Bit- und 32 Bit-Register verwendet. So überlappen einander einige der Pseudoregister implizit. Dies schafft zwei Probleme. Das erste Problem betrifft Optimierungsdurchläufe, die inkorrekte Transformationen vornehmen. Für jede Pseudoregisterdefinition verfolgt der Kompilierer 104 die Instruktionen, die diese Definition verwenden, und für jede Pseudoregisterverwendung verfolgt der Kompilierer 104 ihre Definitionen. Diese Informationen werden use/def-Informationen genannt. Der Kompilierer 104 verwendet use/def-Informationen in Durchläufen wie dem Constant Folding-Durchlauf. Wenn Pseudoregister als Alias füreinander fungieren können, erfordert dies die use/def-Berechnung, und der Kompilierer 104 reicht diese "use" dieser Informationen weiter, um viel komplexer zu sein. Ein zweites Problem, das durch überlappende Pseudoregister geschaffen wird, liegt in der Registerzuordnung. Wenn der Registerzuordner gleichzeitig zwei überlappende Pseudoregister Maschinenregistern zuweist, kann eine Modifikation eines Registers erfordern, daß das andere Register ungültig gemacht wird. Allgemein ist die Verfolgung dieser Informationen sehr schwierig und schafft eine unnötige Komplexität.

Anstatt diese schwierigen Probleme anzugehen und signifikant zur Komplexität des Kompilierers 104 beizutragen, wurde ein Verfahren zum Einfügen spezieller Marker-OASSIGN-Instruktionen ausgebildet, das es dem Kompilierer 104 ermöglichen würde, das Problem zu ignorieren. Ein spezieller Kompiliererdurchlauf unmittelbar nach der IL-Generierung fügt OASSIGNs ein. Nach diesem Kompilierer-104-Durchlauf wird anderen Analysedurchläufen gestattet anzunehmen, daß Pseudoregister einander nicht überlappen (in bezug auf die use/def-Analyse). Ferner ist die Registerzuordnung unter Verwendung von OASSIGNs ziemlich einfach zu behandeln. Wann immer der Registerzuordner zu einem OASSIGN kommt, leert er die Quelle an ihrer Definition und füllt das Ziel nach dem OASSIGN ein. Dieses Verfahren verwendet den Alias-Speicher, um zu garantieren, daß jede Verwendung der Überlappungsdefinition den korrekten Wert verwendet.

Die OASSIGN-Einfügung wird in zwei Stufen behandelt. Zuerst wird eine spezielle Version der UseDef-Analyse laufen gelassen. Diese Version von UseDef kennt Pseudoregisterüberlappungen, und schafft Verwendungslisten und Definitionslisten, die überlappende Pseudoregister enthalten. Der Rest des Kompilierers 104 ist nicht vorbereitet, use/def-Listen zu behandeln, die überlappende Pseudoregister enthalten, daher sollte diese Option für UseDef allgemein nicht verwendet werden. Nachdem diese Analyse vorgenommen wurde, nimmt die Prozedur OOC_Add_Overlap_Defs die tatsächliche Einfügung von OASSIGNs vor. OASSIGN wird für jede Verwendung eingefügt, die eine Überlappungsdefinition aufweist (d. h. eine Definition, die ein Pseudoregister definiert, welches das Pseudoregister der Verwendung überlappt), und für eine Überlappung erreichende Definitionen an Sprungmarken.

Fig. 15 veranschaulicht ein Beispiel eines Falls, wo OASSIGN eingefügt werden würde. In dem Beispiel überlappen einander die Pseudoregister GRPAIR1 und GR1, so daß die Zuweisung zu GRPAIR1 in der ersten Zeile des Codes eine implizite Modifikation von GR1 ist. OASSIGN macht dies explizit.

ANALYSEDURCHLÄUFE

UseDef (compiler/oc_common/src/oc_usedef.c)

Die Berechnung der Verwendungen einer gegebenen Definition und der potentiellen Definitionen für eine gegebene Verwendung ist eine der grundlegendsten Kompilierer-104-Analysen. Jeder Kompilierer-104-Optimierungsdurchlauf

verwendet die use/def-Informationen. Jede der IL-Instruktionen kann ein Pseudoregisterargument, das (in ein Ziel) eingeschrieben wird, und eines oder mehrere Pseudoregisterargumente, die (aus einem src) ausgelesen werden, aufweisen. Nach der UseDef-Analyse hat jedes Ziel eine damit assoziierte Liste, die Zeiger auf alle IL-Instruktionen speichert, welche diesen Wert verwenden könnten (du chain genannt). Ähnlich hat jedes src eine damit assoziierte Liste, die alle IL-Instruktionen speichert, welche diesen Wert definieren (auch ud chain genannt). Das Verfahren zur Berechnung der use/def-Informationen wird nachstehend beschrieben. Es ist ein iteratives Verfahren, das versucht, einen festgelegten Punkt zu erreichen (d. h. bis weitere Iterationen keine Änderungen mehr durchführen).

Die folgenden Schritte wiederholen, bis es zu keiner Änderung der erreichenden Definitionen an irgendeiner Sprungmarke mehr kommt.

Die Definitionsliste für jedes Pseudoregister in regdefs (ein Array von NL-LISTS, indexiert von einem Pseudoregister) löschen.

Über die IL_NODES in statischer Programmreihenfolge iterieren.

Wenn die Instruktion ein Pseudoregister verwendet, die Definition des Pseudoregisters aus regdefs zur ud chain des Operanden kopieren.

Wenn die Instruktion eine Verzweigung ist, die regdefs mit den erreichenden Definitionen kombinieren, die im LABEL der Verzweigung gespeichert sind. Änderungen der erreichenden Definitionen verursachen, daß die gesamte Schleife wiederholt wird.

Wenn die Instruktion ein LABEL ist, die regdefs mit den erreichenden Definitionen bereits am LABEL kombinieren.

Wenn die Instruktion ein Pseudoregister definiert, die Definitionsliste in regdefs so setzen, daß sie nur diese Instruktion enthält.

Wenn die Instruktion eine unbedingte Verzweigung ist, das regdefs Array ändern, damit es der Satz von erreichenden Definitionen ist, die am nächsten LABEL gespeichert sind. Dies wird durchgeführt, da die Instruktionen in ihrer statischen Reihenfolge verarbeitet werden, und die Definitionen, welche die unbedingte Verzweigung erreichen, nicht dieselben sind wie jene, die ihren statischen Nachfolger erreichen.

Live Variable Analysis (compiler/oc_common/src/oc_usedef.c)

Eine weitere Form der Analyse betrifft lebendige variable Informationen. Die lebendige Variablenanalyse wird hauptsächlich zur Registerzuordnung verwendet, kann jedoch auch zur Induktion variabler Transformationen und zur Totcodeeliminierung verwendet werden. Ein Pseudoregister wird als lebendig an einem bestimmten Punkt in einem Programm angesehen, wenn das Pseudoregister entlang einem Ausführungspfad verwendet werden kann, bevor es neu definiert wird. Die lebendige Variablenanalyse markiert auch die letzte Verwendung eines gegebenen Pseudoregisters (eine Verwendung ist die letzte Verwendung, wenn es keine möglichen Ausführungspfade gibt, auf denen das Pseudoregister verwendet wird, bevor es neu definiert wird). Das zur Berechnung der lebendigen variablen Informationen verwendete Grundverfahren wird nachstehend erläutert. Es arbeitet, indem wiederholte Rückwärtsdurchläufe über den Code durchgeführt werden, bis ein festgelegter Punkt erreicht ist.

Die folgenden Schritte wiederholen, bis es zu keiner Änderung mehr der erreichenden Definitionen an irgendeiner Sprungmarke kommt.

Lebendig löschen (ein Bitsatz von Pseudoregistern)

Über die IL_NODES in umgekehrter statischer Programmreihenfolge iterieren.

Wenn die Instruktion ein Pseudoregister verwendet, das Bit des Pseudoregisters lebendig setzen. Wenn das Pseudoregister vorher nicht lebendig war, als letzte Verwendung markieren.

Wenn die Instruktion eine Verzweigung ist, lebendig mit den lebendigen Registern kombinieren, die am LABEL der Verzweigung gespeichert sind. Änderungen an den lebendigen Registern verursachen, daß die gesamte Schleife wiederholt wird.

Wenn eine Instruktion ein LABEL ist, lebendig mit den lebendigen Pseudoregistern bereits an der Sprungmarke kombinieren.

Wenn die Instruktion ein Pseudoregister definiert, Pseudoregister aus lebendig löschen.

Wenn die Instruktion eine unbedingte Verzweigung ist, lebendig löschen. Dies wird durchgeführt, da bei der Verarbeitung der Instruktion in ihrer umgekehrten statischen Reihenfolge die lebendigen Variablen an der unbedingten Verzweigung nicht dieselben sind wie jene an ihrem Nachfolger.

Register Allocation (compiler/oc_common/src/oc_regalloc.c)

Die Registerzuordnung im Kompilierer 104 wird in zwei Stufen durchgeführt. Die erste Stufe nimmt eine Analyse des Codes vor, und bestimmt einen Satz empfohlener Registerzuweisungen auf der Basis eines Modells höherer Ordnung der Zielmaschine. Die zweite Stufe verwendet die Analyse von der ersten Stufe zusammen mit einem weniger abstrakten Maschinenmodell, um den Code tatsächlich zu modifizieren, physische Register zu verwenden. Dieser Abschnitt diskutiert die erste Stufe.

Das Registerzuordnungsverfahren basiert auf der traditionellen Technik der Verwendung der Graphfärbung. Die Knoten des 'Graphen' sind lebendige Pseudoregisterbereiche, mit Kanten zwischen lebendigen Bereichen, die einander überlappen. Eine N-Farbgraphfärbung weist eine von N Farben jedem Knoten zu, so daß keine zwei verbundenen Knoten dieselbe Farbe haben. Wenn der Graph lebendiger Bereiche N gefärbt sein kann (wobei N die Anzahl verfügbarer physischer Register ist), wird klarerweise ein Register jedem lebendigen Bereich zugewiesen. Leider ist die Graphfärbung ein ernstes NP-Problem (d. h. es erfordert exponentielle Zeit), daher wird in der Praxis Heuristik verwendet.

Die Registerzuordnung ist ein komplexes Verfahren mit mehreren Schritten. Die Schritte werden nachstehend detail-

liert beschrieben.

1. Teilen unabhängiger lebendiger Bereiche und Zuordnung von REGINFO-Strukturen

Die Funktion `ComputeRegInfo` tut dies. Sie teilt jedes Pseudoregister in unabhängige lebendige Bereiche, und ordnet eine REGINFO-Struktur für jeden zu. Die REGINFO-Struktur wird verwendet, um Informationen über den betreffenden lebendigen Bereich zu halten, der für die Registerzuordnung verwendet wird, und hält schließlich das 'Zielregister' – das physische Register, das für den lebendigen Bereich zugeordnet wird. Da eine 1 : 1 Entsprechung zwischen lebendigen Pseudoregisterbereichen (ein logisches Konstrukt) und REGINFO-Strukturen besteht, wird der Ausdruck REGINFO oft verwendet, um sich sowohl auf den lebendigen Bereich als auch auf die Datenstruktur zu beziehen.

`ComputeRegInfo` führt das Teilen lebendiger Bereiche nahezu als Nebeneffekt der Zuordnung der REGINFO-Strukturen durch. Die Funktion arbeitet, indem sie mit einer Definition beginnt, die noch keine REGINFO aufweist, eine neue REGINFO dafür schafft, dann rekursiv alle ihre Verwendungen und alle ihre Definitionen (und alle ihre Verwendungen ...) ansieht, und die neue REGINFO mit jeder Definition und Verwendung, die erreichbar ist, assoziiert.

Sobald alle REGINFOS geschaffen worden sind, werden sie in 'einfache' und 'komplexe' geteilt. Eine 'einfache' REGINFO:

Hat exakt eine Definition und eine Verwendung.

Die Verwendung folgt unmittelbar der Definition.

Die Verwendung ist nicht der 2. Operand einer BINOP (zielspezifischen Voraussetzung).

Alle anderen REGINFOS sind komplex. Jede REGINFO erhält eine eindeutige ID, wobei die komplexen im Bereich `[0..c->ri_complex]` liegen, und die einfachen im Bereich `[c->ri_complex..c->ri_total]` liegen. Der Zweck dieser Teilung ist, Speicher beim Halten der Konfliktmatrix zu sparen, die als BITSETs in jeder REGINFO gespeichert ist. Der Effekt der obigen Definition von 'einfach' ist, daß keine zwei einfachen REGINFOS jemals in einen Konflikt miteinander treten können.

2. Berechnung von Konflikten und Kompatibilitäten

Der nächste Schritt ist die Berechnung des Konfliktgraphen der REGINFO-Strukturen. Zwei REGINFOS stehen in einem Konflikt, wenn ihre lebendigen Bereiche überlappen. Zwei REGINFOS sind kompatibel, wenn sie durch Kopieren verbunden werden. In einem Konflikt stehende REGINFOS können nicht demselben Register zugewiesen werden, da sie gleichzeitig lebendig sind. Zwei kompatible REGINFOS sollten demselben Register zugewiesen werden, wenn möglich, da dies eine Kopie eliminiert.

Die Konflikte sind entweder als Graph (mit einem Knoten für jede REGINFO und einer ungerichteten Kante, die jeden REGINFO-Knoten mit jedem anderen Knoten verbindet, mit dem er in Konflikt steht – dies ist die Ansicht, die bei Graphfärbungsverfahren verwendet wird) oder als symmetrische binäre Matrix vorstellbar. Diese letztere Form liegt dem näher, wie die Konflikte tatsächlich gespeichert werden.

Jede REGINFO enthält einen einzelnen BITSET, der eine (ein Teil einer) Reihe der Konfliktmatrix ist. Da keine zwei einfachen REGINFOS miteinander in einem Konflikt stehen können, besteht der untere rechte Quadrant der Matrix nur aus Nullen. Da die Matrix symmetrisch ist, ist der obere rechte Quadrant die Transposition des unteren linken. Folglich muß nur die linke Seite der Matrix gespeichert werden. Daher sind die Konflikt-BITSETs nur jeweils `c->ri_complex`-Bits anstelle von `c->ri_total`.

Zur Bestimmung, ob zwei REGINFOS, A und B, mit den BITSETs in Konflikt stehen, ist es notwendig, zuerst einen Test zu machen, um zu sehen, ob sie einfach oder komplex sind (id mit `c->ri_complex` vergleichen). Wenn eine der beiden komplex ist, das seiner ID entsprechende Bit im Konflikt-BITSET der anderen REGINFO ansehen. Wenn beide komplex sind, kann jedes Bit angesehen werden; sie müssen gleich sein. Wenn keine komplex ist, stehen sie nicht in einem Konflikt.

Konflikte werden aus den Lebendigkeitsinformationen berechnet, die in der IL gespeichert sind (generiert durch `ComputeLive`). `ComputeConflicts` führt einen Einzeldurchlauf über den IL-Code durch, wobei der BITSET komplexer REGINFOS, die am aktuellen Punkt lebendig sind, aus den gesetzten Pseudoregistern, die an diesem Punkt lebendig sind, generiert wird. Während jede komplexe REGINFO zum lebendigen Satz hinzugefügt wird, wird sie markiert, daß sie mit jeder REGINFO, die bereits im lebendigen Satz ist, in Konflikt steht. Beim Auffinden jeder einfachen REGINFO wird sie markiert, daß sie mit dem aktuellen lebendigen Satz in Konflikt steht.

3. Sortieren der REGINFOS nach 'Registerpriorität'

`OC_SortRI` setzt Prioritäten unter den REGINFO-Strukturen auf der Basis verschiedenster abstimmbarer Parameter. Die Gewichtsparameter sind zueinander relativ, daher hat eine Multiplikation aller davon mit einer Konstante keinen Effekt.

`OC_RegAllocConflictWeight`:

Auf die Graphfärbung des Konfliktgraphen gelegtes Gewicht. Höhere Einstellungen dieses Parameters begünstigen Zuordnungen, die mehr verschiedene REGINFOS in Register geben, ungeachtet davon, wie oft diese REGINFOS tatsächlich verwendet werden. Es ist zu beachten, daß REGINFOS mit weniger Verwendungen auch dazu tendieren, eine kürzere Lebenszeit aufzuweisen, und daher gegenüber REGINFOS mit langer Lebenszeit wahrscheinlich begünstigt werden.

`OC_RegAllocDefWeight`:

Auf Definitionen gelegtes Gewicht. Höhere Werte von `OC_RegAllocDefWeight` begünstigen REGINFOS mit mehr verschiedenen Definitions-IL-Anweisungen.

`OC_RegAllocUseWeight`:

Auf Verwendungen gelegtes Gewicht. Sowohl `OC_RegAllocDefWeight` als auch `OC_RegAllocUseWeight` tendieren

dazu, REGINFOs mit langer Lebenszeit und vielen uses/defs zu begünstigen (d.h. nicht REGINFOs, die lange Zeit nur 'herumhängen', ohne verwendet zu werden).

OC_RegAllocResortRat:

Dieser Parameter steuert, wie viel er sortiert, um eine gute Färbung zu erhalten. Wenn OC_RegAllocConflictWeight gleich 0 ist, ist dies irrelevant und sollte 0 sein (= Unendlichkeit). Kleine Zahlen (> 0) bedeuten mehr aufgewendete Zeit und eine bessere Färbung.

4. Registerwahl

Die REGINFOs in einer Serie von Zwangsbedingungen. Die ersten Zwangsbedingungen sind erforderlich, so wird nach ihrer Anwendung, wenn keine Register übrig sind, die REGINFO nicht einem Register zugewiesen (Ziel = -1). Die übrigen Zwangsbedingungen sind erwünscht, aber nicht erforderlich, so wird, wenn irgendeine gegebene Zwangsbedingung dazu führen würde, daß der Satz möglicher Register leer wird, diese übersprungen. Sobald alle Zwangsbedingungen angewendet wurden, wird das Register mit der niedrigsten Nummer aus dem Satz gewählt und dieses verwendet.

TYPE [erforderlich]:
muß ein Register wählen, das einen Wert dieses Typs halten kann (Info von Maschinenmodell).

INUSE [erforderlich]:
kann kein Register wählen, das bereits einer REGINFO zugeordnet wurde, welche in einem Konflikt steht (oder irgend etwas, das diese überlappt).

BASEREFS [erforderlich]:
kann kein Register verwenden, daß der Rahmen als etwas wie frame/stack/base pointer reserviert.

CLOBBED:
es wird versucht, kein Register zu verwenden, daß von jemandem während der Lebenszeit der REGINFO zerstört wird.

DEF CONSTRAINTS:
es wird versucht, ein Register zu verwenden, das die DEST-Zwangsbedingungen vom Maschinenmodell für jede IL erfüllt, die diese REGINFO definiert.

USE CONSTRAINTS:
es wird versucht, ein Register zu verwenden, das die SRC-Zwangsbedingungen vom Maschinenmodell für jede IL erfüllt, die diese REGINFO definiert.

COMPATIBILITY:
es wird versucht, ein Register zu verwenden, das mit einer anderen REGINFO in der Kompatibilitätsliste kompatibel ist, dem bereits ein Register zugewiesen wurde.

Sobald alle REGINFOs Registern zugewiesen wurden (oder dies fehlgeschlagen ist), wird ein erneuter Durchlauf über die REGINFOs durchgeführt, wobei nach über die Kompatibilitätszwangsbedingung zu ändernden Registern gesucht wird (d. h. kompatible REGINFOs, die nach dieser zugewiesen wurden, und die aus irgendeinem anderen Grund nicht in dasselbe Register gehen konnten).

TRANSFORMATIONS-(OPTIMIERUNGS-)DURCHLÄUFE

Die Transformationsdurchläufe befinden sich im Kern des optimierenden Kompilierers 104. Jeder Durchlauf macht einen Versuch, einen Teil des Codes neu zu schreiben, so daß die Bedeutung des Codes gleich bleibt, der produzierte Endcode jedoch schneller läuft. Einige der Transformationsdurchläufe verbessern selbst nicht die Qualität des Codes, sondern ermöglichen stattdessen anderen Durchläufen, den Code zu verbessern. So tendieren die Durchläufe dazu, am besten in Kombinationen zu arbeiten, und sind weniger effektiv, wenn sie allein verwendet werden. Daher werden viele Durchläufe wie Dead Code Elimination wiederholt laufen gelassen.

Dead Code Elimination (compiler/oc_common/src/oc_usedef.c)

Der Totcode-Eliminierungsdurchlauf (OC_ElimDeadCode) entfernt jeden Code, der tot ist, sowohl auf der Basis der Datenfluß- als auch der Steuerflußinformationen. Datenflußinformationen werden verwendet, um IL_NODES zu eliminieren, die keine Nebeneffekte haben, und deren Ergebnisse ungenutzt sind. Steuerflußinformationen werden zur Entfernung aller IL_NODES verwendet, die niemals ausgeführt werden (unerreichbarer Code). Ferner wird eine bestimmte neue Verzweigungszielsetzung vorgenommen. Das verwendete Verfahren wird nachstehend beschrieben.

Die folgenden Schritte wiederholen, bis keine Änderungen mehr gemacht werden.

1. Über die IL_NODES in statischer Programmreihenfolge iterieren.

a) Wenn die Instruktion unerreichbar ist, diese entfernen. Die Instruktion ist unerreichbar, wenn sie ein LABEL ist, das nicht das Ziel irgendeiner anderen Instruktion ist, oder wenn sie ein GOTO oder CGOTO zur nächsten Instruktion ist, oder wenn die Instruktion direkt nach einer unbedingten Verzweigung steht und kein LABEL ist.

b) Wenn die Instruktion keinen Nebeneffekt hat, und sie keine andere Verwendung als sich selbst hat, diese entfernen.

c) Wenn eine festgelegte Verzweigungsinstruktion zu einer unbedingten Verzweigung springt, die Instruktion neu zielen (z. B. ein GOTO zu einem GOTO).

d) Auf eine bedingte Verzweigung zur nächsten Instruktion prüfen, die von einer Verzweigung nach anderswo gefolgt wird (L2). In diesem Fall wird die Bedingung umgekehrt, und die bedingte Verzweigung wird neu auf L2 zielgerichtet.

Fig. 16 veranschaulicht insbesondere ein Beispiel einer Totcodeeliminierung und Adressenprüfungseliminierung.

(compiler/ooct_elim_achk.c)

Der Durchlauf zur Adressenprüfungseliminierung verwendet Datenflußanalysetechniken, um unnötige Adressenausrichtungsprüfungen zu eliminieren. Der Code arbeitet mittels der Durchführung einer Wertinferenz über eine Algebra von gerade und ungerade. Mit anderen Worten wird der Code analysiert, um zu bestimmen, ob an irgendeinem gegebenen Punkt ein Pseudoregister einen geraden, ungeraden oder unbekannten Wert enthält. Diese Analyse wird global durchgeführt und arbeitet quer über Verzweigungen. Dies bedeutet, daß sie für Schleifen und durch andere Steuerflüsse arbeitet, und besonders gut arbeitet, wenn eine einfache Aufrollung von Schleifen vorgenommen wird.⁴ Das verwendete Verfahren wird nachstehend beschrieben. Es ist ein iteratives Verfahren, das versucht, einen konservativen festgelegten Punkt zu erreichen. Werte werden auf drei Hauptwegen abgeleitet. Erstens kann, wenn ein Pseudoregister einer Konstanten zugewiesen wird, der Wert abgeleitet werden. Zweitens kann, wenn ein Pseudoregister das Resultat einer Operation mit bekannten Argumenten ist, der Wert abgeleitet werden. Beispielsweise werden zwei gerade Zahlen addiert, um eine weitere gerade Zahl zu ergeben. Schließlich liefern bedingte Verzweigungen Informationen über den Wert von Pseudoregistern. Wenn beispielsweise ein Pseudoregister auf den geraden Zustand getestet wird, wissen wir, daß es entlang einer Verzweigung gerade ist, und daß es entlang der anderen Verzweigung ungerade ist.

Die folgenden Schritte wiederholen, bis es zu keiner Änderung der überlagerten (interferenced) Werte an irgendeiner Sprungmarke mehr kommt.

1. Die Definitionsliste für jedes Pseudoregister in `invals` löschen (ein Array von `INVALs`, indexiert von einem Pseudoregister).
2. Über die `IL_NODES` in statischer Programmreihenfolge iterieren.
 - a) Wenn die Instruktion angesichts der aktuell bekannten Inferenzwerte vereinfacht werden kann, die Instruktion durch die einfachere Version ersetzen. Änderungen der Instruktion verursachen, daß die gesamte Schleife wiederholt wird.
 - b) Die `invals` auf der Basis der Ausführung der aktuellen Instruktion aktualisieren.
 - i) Wenn die Instruktion bedingt ist, von der ein Wert abgeleitet werden kann, die am Ziel-LABEL und CGOTO gespeicherten Inferenzwerte mit dem geeigneten Inferenzwert aktualisieren.
 - ii) Wenn die Instruktion unbedingt ist und ein Pseudoregister definiert, den Wert dieses Pseudoregisters in `invals` aktualisieren. Der Wert ist unbekannt, außer die Operation ist ein SET, oder ist ein Spezialfall, wie die Addition von zwei geraden Zahlen.
 - c) Wenn die Instruktion ein LABEL ist, die `invals` mit den Inferenzwerten bereits an der Sprungmarke kombinieren.
 - d) Wenn die Instruktion eine Verzweigung ist, die `invals` mit den an der Sprungmarke der Verzweigung gespeicherten Inferenzwerten kombinieren. Änderungen der `invals` verursachen, daß die gesamte Schleife wiederholt wird.
 - e) Wenn die Instruktion eine bedingte Verzweigung ist, werden alle von dieser Bedingung abgeleiteten Werte mit `invals` kombiniert.
 - f) Wenn die Instruktion eine unbedingte Verzweigung ist, das `invals`-Array ändern, um die am nächsten LABEL gespeicherten Inferenzwerte zu werden. Dies wird durchgeführt, um die Instruktionen in ihrer statischen Reihenfolge zu verarbeiten, und die abgeleiteten Werte an der unbedingten Verzweigung sind nicht dieselben wie jene an ihrem statischen Nachfolger.

Fig. 17 veranschaulicht insbesondere ein Beispiel der Adressenprüfungseliminierung. Um die Leistung der Analyse zu verbessern, kann ein Pseudoregister andere Werte annehmen als einfach ODD, EVEN oder UNKNOWN. Ein Pseudoregister kann auch als EQUIVALENT zu einem anderen Pseudoregister oder EQUIVALENT zu einer binären Operation von zwei Pseudoregistern markiert werden. Dies verbessert die Qualität der Analyse, indem ermöglicht wird, daß Informationen über ein Pseudoregister zu anderen Pseudoregistern propagiert werden. Beispielsweise wird angenommen, daß das Pseudoregister R1 und das Pseudoregister R2 für äquivalent befunden werden. Wenn das Verfahren zeigen kann, daß R1 gerade ist (beispielsweise über ein Verzweigungstestergebnis), muß auch R2 gerade sein.

Es ist zu beachten, daß das Verfahren ein konservatives ist, die Werte, die abgeleitet werden, müssen monoton ansteigen. Mit anderen Worten, wenn zu irgendeiner Zeit während der Ausführung das Verfahren bestimmt, daß ein Wert an einem Punkt im Programm EVEN ist, muß es der Fall sein, daß der Wert wirklich EVEN ist. Das Verfahren zeigt niemals an, daß ein Pseudoregister während einer Iteration EVEN ist, und daß es während einer anderen Iteration UNKNOWN ist. Aus dieser Eigenschaft kann geradlinig die Beendigung des Verfahrens abgeleitet werden.

Hoisting (compiler/oc_common/src/oc_hoist.c)

Hoisting, das allgemein als schleifeninvariante Codebewegung bezeichnet wird, ist der Prozeß der Bewegung von Berechnungen, die in bezug auf eine Schleife außerhalb dieser Schleife konstant sind. Dies sieht allgemein eine Beschleunigung vor, da der Code nur ein einziges Mal anstatt von einmal für jede Schleifeniteration ausgeführt wird.

1. IL neu numerieren (d. h. so daß die id-s in einer Reihenfolge sind).
2. Für jede Rückwärtsverzweigung (d. h. eine potentielle Schleife) Dinge auszuhoisten versuchen.
 - a) Wenn es einen weiteren Einsprungpunkt in die Schleife gibt, wird nichts aus dieser Schleife ausgehoistet.
 - b) Über die `IL_NODES` innerhalb der Schleife in statischer Reihenfolge iterieren,
 - i) Wenn ein Knoten die folgenden Bedingungen erfüllt, kann er gehoistet werden:

- (a) Er verwendet oder definiert kein 'echtes Register'.
- (b) Er verwendet kein innerhalb der Schleife gesetztes Pseudoregister.
- (c) Er hat keine Nebeneffekte.
- ii) Für irgendeine OP, die gehoistet werden kann, jedes Pseudoregister, die sie definiert, neu benennen.
- iii) Den IL_NODE über die Schleife bewegen.
- iv) Alle IL_NODES neu nummerieren.
- v) Wenn eine Verzweigung detektiert wird, zum Ziel der Verzweigung überspringen (da nicht bestimmt werden kann, ob die Verzweigung ausgeführt wird, kann der Code nicht gehoistet werden).

Der Hoisting-Durchlauf ist für den OOC nicht immer effektiv. Der Hauptgrund dafür ist, daß viele Schleifen auch Einsprungpunkte sind, so daß sie mehrfache Einsprünge in die Schleife aufweisen und vom Hoisting-Durchlauf nicht berücksichtigt werden. Dieses Problem könnte mittels der Durchführung einer "Sprungmarkenteilung" behoben werden, wobei eine neue Sprungmarke geschaffen wird, die als Ziel für die Schleife verwendet wird. Gehoistete Operationen können dann zwischen die Originalsprungmarke und die neu geschaffene Sprungmarke gehoben werden. Dies wird bald implementiert.

Common Subexpression Elimination (CSE) (compiler/oc_common/src/oc.cse.c)

Common Subexpression Elimination (CSE) ist eine Technik, die zur Eliminierung redundanter Berechnungen dient. Der Kompilierer 104 verwendet ein globales CSE-Verfahren. Das Grundverfahren wird nachstehend zusammen mit einem Erläuterungsbeispiel in Fig. 18 beschrieben.

1. Während der Durchführung von Änderungen für jede IL_NODE, die ein Ziel aufweist (Zeile 1 im Beispiel) folgendes tun:
 - a) Paarweise alle Verwendungen des Ziels prüfen, um zu sehen, ob eine die andere dominiert (A dominiert B, wenn alle Pfade zu B durch A gehen müssen). Für jedes derartige Paar A und B (Zeile 2 und 4) folgendes tun:
 - ii) Prüfen, ob A und B 'übereinstimmen' (gleicher OP-Code und gleiche Quellen), wenn nicht, zum nächsten Paar von Ausdrücken gehen. A und B sind ein 'allgemeiner Teilausdruck'.
 - iii) Ausgehend von A und B auf folgende Weise einen größeren allgemeinen Teilausdruck zu finden versuchen. Wenn A und B Ziele aufweisen, und das Ziel von B eine eindeutige Verwendung C hat (Zeile 5), prüfen, ob das Ziel von A irgendeine Verwendung D hat (Zeile 3), so daß D C dominiert, und D mit C übereinstimmt. Wenn dies der Fall ist, D und C zum allgemeinen Teilausdruck hinzufügen, und einen größeren Teilausdruck mit A=D, B=C zu finden versuchen.
 - iv) Da nun zwei allgemeine Klammerausdrücke A (Zeilen 2, 3) und B (Zeilen 4, 5) vorliegen, muß der Code neu geschrieben werden, so daß die Verwendungen von B nun statt dessen A verwenden. Wenn das Ziel von A vor der Verwendung durch B geändert werden könnte, wird eine Kopie in ein neues Pseudoregister verwendet.

Fig. 18 veranschaulicht insbesondere ein Beispiel der Common Subexpression Elimination (CSE).

Copy Propagation (compiler/oc_common/src/oc_copyprop.c)

Copy Propagation ist eine Transformation, die versucht, Verwendungen des Ziels einer Zuweisung durch die Quelle der Zuweisung zu ersetzen. Während Copy Propagation selbst nicht die Qualität des Codes verbessert, wird häufig ein Code produziert, wo das Ergebnis einer Zuweisung nicht länger verwendet wird, und so kann die Zuweisung eliminiert werden. Das Copy Propagation-Verfahren wird nachstehend beschrieben.

1. Für jede ASSIGN-Operation.

- a) Wenn die Quelle von ASSIGN eine einzige Definition aufweist, und die einzige Verwendung dieser Definition ASSIGN ist, und das Ziel von ASSIGN zwischen der Definition und ASSIGN weder modifiziert noch verwendet wird, dann die Definition in eine Definition für das Ziel von ASSIGN modifizieren und ASSIGN entfernen.
- b) Für jede Verwendung des Ziels von ASSIGN testen, ob ASSIGN die einzige Definition dieser Verwendung ist, und testen, ob die ASSIGN-Quelle zwischen ASSIGN und der Verwendung sowohl lebendig als auch gültig ist. Wenn beide Tests wahr sind, die Verwendung des Ziels durch eine Verwendung der Quelle ersetzen.

Fig. 19 veranschaulicht insbesondere ein Beispiel einer Copy Propagation. Fig. 20 veranschaulicht insbesondere ein Beispiel einer Constant Folding.

Constant Folding (compiler/oc_common/src/oc_cfold.c)

Constant Folding ist eine Transformation, die Operationen an konstanten Werten zur Kompilierzeit untersucht. Wenn die IL zwei Konstanten miteinander addiert, ersetzt Constant Folding beispielsweise diese IL-Instruktionen durch eine einzige SET-Instruktion, die das Ziel der Addition der Summe der beiden Konstanten zuweist.

Das Verfahren für den Constant Folding-Durchlauf ist sehr geradlinig. Jede IL-Instruktion wird der Reihe nach untersucht. Für jede arithmetische und logische Operation (ADD, SUB, BAND, BOR etc.) wird, wenn alle ihre Argumente Konstanten sind, die IL-Operation durch eine SET-Operation ersetzt, die das Ziel im Pseudoregister auf den Wert der Operation an den konstanten Argumenten setzt.

Der Kompilierer 104 hat auch einen Musterübereinstimmungs- bzw. engl. Pattern Matching-Optimierungsdurchlauf, der bekannte Muster von IL-Instruktionen durch effizientere Versionen ersetzt. Es gibt derzeit keine Muster, die allgemein mit vom OOCT generierten IL-Mustern übereinstimmen, so daß der Pattern Matching-Durchlauf nicht durchgeführt wird.

ZIELCODEGENERIERUNG

Nachdem die IL generiert wurde, und die Transformationen durchgeführt wurden, um die Qualität des Codes zu verbessern, werden drei Kompilierer-104-Hauptdurchläufe verwendet, um den Code zu generieren. Bis zu diesem Punkt waren die IL und die Transformationsdurchläufe maschinenunabhängig, diese drei Durchläufe sind jedoch stark von der Zielarchitektur abhängig.

INSTRUCTION FOLDING (compiler/oc_common/src/ix86_ifold.c)

Die OOCT-IL ist eine RISC-ähnliche Architektur, die ohne Modifikation nicht effizient auf die Zielarchitektur abgebildet wird. Insbesondere wäre es suboptimal, eine Zielinstruktion für jede IL-Instruktion zu emittieren. Da die Zielarchitektur eine CISC-Architektur ist, können oft mehrfache IL-Instruktionen zu einer einzigen Zielinstruktion kombiniert werden. Der Instruktionsfaltungs- bzw. engl. Instruction Folding-Durchlauf ist ausgebildet, um dieses Problem zu lösen, indem Gruppen von IL-Instruktionen markiert werden, die zu einer einzigen Zielinstruktion kombiniert werden können.

Der Instruction Folding-Durchlauf arbeitet, indem nach einer von einer Anzahl verschiedener vordefinierte Instruktionsskombinationen gesucht wird. Die folgenden Kombinationen werden verwendet:

- Konstanten werden in verschiedene Operationen gefaltet, wie ADD, SUB etc.
- SETCC-Instruktionen werden in die Instruktion gefaltet, auf deren Basis sie die BedingungsCodes setzen.
- DIV-, REM-Paare mit denselben Argumenten werden zusammen gefaltet.
- ADD-, SUB- und ASL-Operationen können in eine einzige 'lea'-Operation oder in die Adressenberechnung von LOAD oder STORE gefaltet werden.
- 16 Bit-BSWAP-, STORE-Kombinationen werden in zwei getrennte 8 Bit-Speicherungen gefaltet.
- LOAD-Operationen werden in verschiedene Operationen gefaltet, wenn ihr Ergebnis als zweites Argument verwendet wird.

Der Instruction Folding-Durchlauf entscheidet einfach, wenn Instruktionen gefaltet werden sollten, er führt die tatsächliche Faltung nicht aus, die dem Maschinencode-Generierungsdurchlauf überlassen wird. Der Instruction Folding-Durchlauf markiert zu faltende Instruktionen auf zwei Wegen. Erstens kann jeder Operand eines Knotens mit einem "fold"-Bit markiert werden. Zweitens werden Instruktionen, von denen alle Verwendungen in eine andere Instruktion gefaltet sind, mit einer IL_COMBINE-Flagge und mit dem mmFold-Feld markiert, das Informationen über den Weg liefert, auf dem die Instruktion gefaltet wird. Der Registerzuordner und die Maschinencodegenerierung verwenden diese Felder, um korrekt zu arbeiten.

Target REGISTER ALLOCATION (compiler/oc_common/src/ix86_regalloc.c)

Sobald der Registerzuordner (RegAlloc) Register für alle REGINFOs gewählt hat, die er wählen kann, ist es notwendig, den Code durchzugehen und ihn zu modifizieren, um diese physischen Register anstelle der Pseudoregister zu verwenden. Ferner ist es notwendig, einige zusätzliche Pseudoregister temporär in reale Register zu geben, so daß der Assembler den Code für diese Instruktionen generieren kann. Dies wird im allgemeinen das Einfügen eines Leer- und Füllcodes erfordern, um die Werte zu speichern und wiederherzustellen, die der RegAlloc in diese Register plaziert hat. Um dies durchzuführen, verwendet OC_RegUseAlloc einen Zwangszuordner (GetReg) und fügt Leer- und Füllzeichen ein, um Register erneut zu verwenden.

OC_RegUseAlloc führt einen einzelnen Durchlauf über den Code durch, wobei der Zustand der physischen Register in einem 'stat'-Array modifiziert und verfolgt wird. Das stat-Array zeichnet auf, was in jedem Register zu einem gegebenen Moment ist (oder sein sollte), und ob der Wert im Register oder die Leerstelle (oder beide) korrekt ist. OC_RegUseAlloc arbeitet als Serie von Stufen, von denen jede spezifische Modifikationen an den aktuell verarbeiteten Instruktionen vornimmt. Wenn mehrfache IL-Instruktion durch den Instruction Folding-Durchlauf zusammen gefaltet wurden, werden sie als einzige Instruktion behandelt. Die Stufen sind wie folgt:

1. Wenn die Instruktion irgendein physisches Register direkt verwendet, sicherstellen, daß jegliche Füllzeichen in diesen Registern nach dieser Verwendung auftreten. Die Instruktion modifizieren, um Register zu verwenden, die den Pseudoregistern durch die RegAlloc-Analyse zugeordnet wurden. Alle Register verriegeln, so daß sie nicht erneut verwendet werden.
2. Die Instruktion modifizieren, um Register zu verwenden, die durch GetReg-Aufrufe der vorhergehenden Instruktion temporären zugeordnet wurden. Alle diese Register verriegeln.
3. Die Statusinformationen im stat-Array säubern, um alle Register zu reflektieren, welche die Instruktion zerstört, wobei Leerzeichen nach Bedarf eingefügt werden. Das Zielregister in das Register ändern, das vom RegAlloc zugeordnet wurde, sofern dies geschehen ist (es ist zu beachten, daß es nicht notwendig ist, dieses Register zu verriegeln, da es verwendet werden kann, um ein src zu halten, wenn notwendig).
4. Den Code modifizieren, um Quellen in das Register zu geben, wo dies für die Zielcodegenerierung notwendig

ist. Dies involviert einen Aufruf von GetReg für jene Quellenoperanden, die in Registern sein müssen.

5. Alle Register, die verriegelt wurden, freigeben.

6. Ziele festlegen, um reale Register zu verwenden, wo dies für den Zielcode notwendig ist. Dies involviert einen Aufruf von GetReg.

7. Das stat-Array finalisieren, um das Ergebnis dieser Operation zu reflektieren, und alle verwendeten Register festlegen, wobei ihre 'before'-Stellen in die nächste Instruktion gesetzt werden (so daß jegliche Leer/Füllzeichen nach dieser vollendeten Instruktion platziert werden).

Es ist wichtig, daß stat-Array zu verstehen. Es ist ein Array von Datenstrukturen, die von einem physischen Register (alle Register unter MM_NumRegs sind physische Register) indexiert werden, das den Status eines gegebenen physischen Registers anzeigt. Die Struktur enthält die folgenden Felder:

1. ri: Die REGINFO-Struktur, die das Pseudoregister identifiziert, das aktuell mit diesem realen Register assoziiert ist (kann 0 sein, um keine Assoziation anzuzeigen). Dies kann entweder ein Pseudoregister, das diesem Register vom RegAlloc zugeordnet wurde, oder ein von GetReg temporär zugewiesenes sein.

2. alt_ri: Eine REGINFO-Struktur, die ein zusätzliches Pseudoregister identifiziert, das auch in diesem Register ist. Dies wird verwendet, wenn GetReg ein Pseudoregister einem physischen Register zuweist, während der RegAlloc ein anderes hierher (in ri) gegeben hat.

3. flags: Flaggen zum Identifizieren des Zustands des Registers. RegValid wird beispielsweise verwendet, um anzuzeigen, daß der Wert im Register gültig ist. Wenn RegValid nicht gesetzt ist, muß das Register gefüllt werden, bevor es verwendet werden kann. Für eine vollständige Beschreibung der möglichen Flaggen siehe ix86_regalloc.

4. before: Die Instruktion, wo Leer- oder Füllzeichen für dieses Register platziert werden sollten.

MASCHINENCODEGENERIERUNG

Der Maschinencode für das Ziel wird in zwei Durchläufen generiert. Der erste Durchlauf wird zur Bestimmung der Größe der Instruktionen verwendet, so daß Verzweigungsdistancen berechnet werden können. Der zweite Durchlauf nimmt die tatsächliche Codegenerierung vor. Die beiden Durchläufe sind identisch, außer daß der erste den Code in einen Arbeitspuffer generiert, und nicht die korrekten Verzweigungsdistancen aufweist, so daß nahezu der gesamte Code gemeinsam genutzt wird.

Beide Durchläufe bestehen aus einem Einzeldurchlauf durch die IL-Instruktionen der Reihe nach. Für jede Instruktion wird eine durch den OP-Code und Typ indexierte Tabelle verwendet, um eine Funktion zur Generierung des Codes wiederzufinden. Diese Codegenerierungsfunktionen verwenden EMIT-Makros, die ein generalisiertes Verfahren zur Generierung von Zielinstruktionen sind, ohne daß die genaueren Details des Ziels bekannt sein müssen (siehe ix86_Asm_Emit.[h,c]). Diese Makros vereinfachen die Assemblierung von Instruktionen, die irgendeinen der Zieladressierungsmodi verwenden.

SEGMENTVERWALTUNG

Der vom OOCOT kompilierte Code wird innerhalb einer SEGMENT-Datenstruktur gespeichert. Es gibt zahlreiche wichtige Themen, die mit der Verwaltung von Segmenten assoziiert sind. Erstens haben Segmente einen speziellen Speicherzuordner, um die Segmentspeicherung zu behandeln. Zweitens wird diskutiert, wie Segmente geschaffen und in das System installiert werden. Drittens wird diskutiert, wie Segmente gelöscht werden (wenn diese Option eingeschaltet ist).

Schließlich wird die Segmentverriegelung diskutiert, wenn die Segmentlöschung ein ist.

SEGMENT ALLOCATOR (compiler/SegAlloc.[h,c])

Die Speicherverwaltung für Segmente im OOCOT wird mit einem speziellen Zuordner bearbeitet. Zur Zeit der OOCOT-Initialisierung wird der Segmentzuordner bzw. engl. Segment Allocator (SegAlloc) mit einem großen Speicherstück initialisiert. Dann sieht die SegAlloc-Einheit die Fähigkeit vor, ein ungenutztes Speicherstück mit variabler Größe (wie malloc) anzufordern, um ein vorher zugeordnetes Speicherstück frei zu machen (wie free), und eine Statistik über die aktuelle Speicherverwendung anzufordern.

Der SegAlloc ist komplexer als der ZONE-Zuordner, da er eine variable Größenzuordnung behandeln muß. Der SegAlloc verwendet ein ziemlich standardmäßiges Zuordnungsverfahren. Der Zuordner hält eine sortierte freie Liste von Stücken und verwendet einen 32 Bit-Anfangsblock für zugeordnete Blöcke, um ihre Größe anzuzeigen. Um ein Speicherstück zuzuordnen, wird die freie Liste nach einem Stück durchsucht, das zur angeforderten Größe paßt. Wenn der Rest des Stücks größer ist als eine Mindestgröße, wird er geteilt, und der Rest wird zur freien Liste hinzugefügt. Um ein Stück frei zu machen, wird es zur freien Liste hinzugefügt. Da die Geschwindigkeit des Freimachens von Speicher kein kritischer Faktor ist, wird die freie Liste nach benachbarten freien Blöcken durchsucht, die zu einem einzelnen freien Block kombiniert werden.

SEGMENTSCHAFFUNG UND -INSTALLATION (compiler/oocot_trace.c, compiler/SegMgr.[h,c],)

Nachdem die Hauptstufen der Kompilierung vollendet sind, ist das Endergebnis ein Speicherblock, der den verschieblichen Zielcode enthält. Der nächste Schritt ist die Schaffung eines Segments für diesen Code, und die Installation dieses Segments in den für Segmente zugeordneten Raum. OOCOT_Install nimmt diese Funktion vor. Anfänglich wird Platz für das Segment in der ZONE-Speicherregion zugeordnet. Das Segment wird mit einer Liste der Basisblöcke, die vom

Blockwähler 114 gewählt werden, daß die Segmente später gesucht werden können, herauszufinden, ob sie eine gegebene Originalinstruktion enthalten), und mit dem generierten Code initialisiert. Ein Aufruf von SEGMMGR_Install macht das Segment zu einem kontinuierlichen Speicherblock und kopiert diesen in den Raum, der unter Verwendung der SegAlloc-Einheit für Segmente zugeordnet wurde.

Nachdem das Segment geschaffen und in den Segmentzuordnungsraum bewegt wurde, muß die Übersetzungstabelle, die anzeigt, welche Originalinstruktionen für sie kompilierten Code haben, aktualisiert werden. Für jede der Originalinstruktionen, die externe Einträge sind, wird die Übersetzungstabelle mit der korrekten Adresse im für diesen Eintrag generierten Code aktualisiert. Ferner wird die Übersetzungstabelle mit der TRANS_ENTRY_FLAG markiert, um anzuzeigen, daß die K-Instruktion einen gültigen Eintrag hat.

SEGMENTLÖSCHUNG (compiler/ooot_trace.c, compiler/SegDel.[h,c])

Wenn der Kompilierer 104 einen Eintrag in die Übersetzungstabelle schreibt, kann er einen alten überschreiben, der sich bereits darin befand. Kein Interpretierer 110 wird den alten Eintrag lesen und zum alten Segment springen können. Wenn ein Segment keine Einträge in der Übersetzungstabelle aufweist, und es keinen Interpretierer 110 gibt, der das Segment verwendet, kann es gelöscht werden, und sein Speicher kann für ein anderes Segment verwendet werden. Dieser Abschnitt beschreibt, wie der Kompilierer 104 detektiert, daß ein Segment gelöscht werden kann, und dieses dann löscht. Der Kommunikationsabschnitt beschreibt auch detailliert die Segmentverriegelung und die Segmentlöschung.

Wenn der Kompilierer 104 einen Einsprungpunkt in der Übersetzungstabelle überschreibt, stellt er den alten Einsprungpunkt auf eine Löschliste. Nach der Installation eines neuen Segments ruft der Kompilierer 104 SEGDEL_TryDeletions auf. Diese Prozedur prüft jeden Eintrag auf der Löschliste. Wenn kein Interpretierer einen Einsprungpunkt verwendet, wird er gelöscht, so daß er später erneut verwendet werden kann.

Jedes Segment hat darin einen Einsprungpunktzähler. Wenn ein Einsprungpunkt gelöscht wird, verringert der Kompilierer 104 den Einsprungpunktzähler um das Segment, das ihn enthält. Wenn der Einsprungpunktzähler eines Segments 0 erreicht, verwendet kein Interpretierer 110 das Segment, und kein neuer Interpretierer 110 kann in dieses springen. Der Kompilierer 104 löscht das Segment und macht seinen Speicher für eine Verwendung durch andere Segmente frei.

SEGMENTVERRIEGELUNG

Jeder Einsprungpunkt in ein Segment hat einen Zähler, der als Verriegelung für den Einsprungpunkt arbeitet. Der Zähler zeichnet die Anzahl von Interpretierern 101 auf, die den Einsprungpunkt verwenden. Wenn der Zähler größer als Null ist, werden der Einsprungpunkt und sein Segment verriegelt, und der Kompilierer 104 löscht diese nicht. Das wichtigste Merkmal der Einsprungpunktverriegelung ist, daß die Instruktionen, die das Segment verriegeln und entriegeln, nicht Teil des Segments selbst sind. Dadurch wird es dem Interpretierer 110 ermöglicht, beliebige Instruktionen im Segment auszuführen, außer es hält die Verriegelung. Die Dokumentation für den Kompilierer 104 und Interpretierer 110 erläutert detailliert den Segmentverriegelungsmechanismus.

ANDERE THEMEN

Es gibt zahlreiche andere Themen betreffend den Kompilierer 104, die nicht gut in andere Abschnitte passen, jedoch wichtig zu verstehen sind.

STACK WARPING (common/ooot_warp.[c,h])

Der Kompilierer 104 wird anfänglich einem kleinen Stapel zugeordnet, der nicht dynamisch expandiert. Da der Kompilierer 104 zahlreiche rekursive Prozeduren verwendet, ist leider oft die Größe des Stapels, den er benötigt, größer als die vorgesehene. Bei auf GranPower laufenden Programmen wurden Situationen beobachtet, in denen Seitenfehler vom Kompilierer 104 aufgrund eines Stapelüberlaufs nicht behoben werden konnten. Anstatt zu versuchen, Abschnitte des Kompilierers 104 neu zu schreiben, oder zu bestimmen, wie Seitenfehler aufgrund eines Stapelüberlaufs korrekt zu behandeln sind, wird ein viel größerer Stapel verwendet als jener, der vom OOOT_buffer zugeordnet wurde. Die Größe dieses Stapels wurde so gewählt, daß die Stapelgröße niemals ein einschränkender Faktor sein würde (andere Faktoren, wie die ZONE-Größe, sind eine größere Einschränkung). Um diesen Stapel zu verwenden, wurde eine saubere Schnittstelle ausgebildet, OOOT_Warp_Stack, die es ermöglicht, daß eine Funktion unter Verwendung des großen Stapelraums des OOOT aufgerufen wird. Beim Rücksprung von OOOT_Warp_Stack bleibt der Stapelzeiger unverändert. Wenn in den Kompilierer 104 über ooot_Compile_Seed eingesprungen wird, dem Haupteinsprungpunkt, um einen Startparameter zu kompilieren, wird er daher unter Verwendung von OOOT_Warp_Stack aufgerufen.

ASSERTIONEN (common/assert.[c,h])

Der Code im Kompilierer 104 hat eine große Anzahl von Assertionsanweisungen. Assertionen werden im gesamten Kompilierer 104 verwendet, um Konsistenz einschränkungen zu prüfen, und für andere Fehlerzustände. Assertionen spielen zwei wichtige Rollen. In der Diagnoseumgebung verursacht ein Assertionsausfall, daß das Programm anhält, während zur Verfolgung des Problems nützliche Informationen angezeigt oder gespeichert werden. In der Produktionsumgebung werden Assertionen verwendet, um Fehlerzustände einzufangen, und für einen sicheren Ausgang aus der Kompilierung, wenn diese Zustände auftreten. Wenn der Kompilierer 104 beispielsweise über keinen Speicher mehr verfügt, verursacht eine Assertion, daß der Kompilierer 104 die Kompilierung dieses Startparameters abbricht.

Die Serviceeinheit sieht Dienste vor, welche in Standard-C-Bibliotheken typischerweise vorgesehen werden, wie printf und memset, die vom KOI-Monitor nicht vorgesehen werden. Diese Einheit soll die Notwendigkeit wegabstrahieren, diese Systemaufrufe im Windows- und Firmware-Aufbau anders zu behandeln. Es gibt zwei grundlegende Implementationen dieser Serviceroutinen, eine für das Wintest-Projekt und die andere für den Firmware-Aufbau.

VIII. WINDOWS-TESTUMGEBUNG

Die Windows-Testumgebung spielt eine wesentliche Rolle bei der raschen Entwicklung und den Tests des OOCT-Systems. Durch die Entwicklung unter Windows werden Standard-Diagnosewerkzeuge unter MSVC vorgesehen. Ferner sind nützliche Werkzeuge wie Profiler verfügbar. Zu Testzwecken wurden spezielle Testverfahren unter Windows entwickelt, welche die Testgeschwindigkeit und den Testumfang erweitert haben.

Zuerst wird die simulierte Granpower-Umgebung beschrieben. Dann wird die Vergleichseinheit diskutiert, welche die meisten der fortgeschrittenen Testtechniken vornimmt.

Schließlich werden Codeauszüge des Kompilierers 104 beschrieben.

SIMULIERTE GRANPOWER-UMGEBUNG

Zur Durchführung der anfänglichen Tests des OOCT sowie der fortgeschrittenen Tests und der Leistungsanalyse war ein Interpretierer notwendig, der unter Windows laufen würde. Der Interpretierer 110 selbst erforderte keine Modifikationen, aber Initialisierungsaufrufe und AOI-Systemaufrufe, die auf dem GranPower-System zur Verfügung stehen, mußten geschrieben werden. Ferner war, damit der OOCT unter Windows läuft, eine Ausbildung notwendig, um mehrfache "Aufgaben" laufen zu lassen, da der Kompilierer 104 als getrennte Aufgabe vom Interpretierer 110 läuft.

INITIALISIERUNG

Der erste Teil der Schaffung einer simulierten Umgebung unter Windows war die Schaffung eines Codes zur korrekten Initialisierung von KOI-Datenstrukturen und zur Simulation der KOI-Initialisierungs-API für die OOCT-Aufgabe. Der Interpretierer 110 erwartet, daß eine Anzahl von Datenstrukturen geeignet initialisiert wird, um irgendeinen Code auszuführen. Ferner steuern bestimmte Datenstrukturelemente, ob der OOCT verwendet wird. Indem unser Initialisierungsscode auf dem Firmware-Initialisierungsprozeß basiert, Simulation der korrekten Initialisierung, um den Interpretierer 110 laufen zu lassen, und einen Teil seines Grundverhaltens zu steuern. Ähnlich wurde grundsätzlich eingerichtet, daß die KOI-Initialisierungs-API für die OOCT-Aufgabe auf dem von der Firmware verwendeten Code läuft. Dadurch wurde ermöglicht, daß das anfängliche Schreiben und Testen von Schnittstellen zwischen dem Interpretierer 110 (wie Aufrufe von OOCT_Init) unter Standard-Windows-Diagnoseumgebungen arbeitet. Es wurde auch geradlinig, die Schnittstelle zu ändern und zu testen.

AOI-SYSTEMAUFRUFE (wintest/MiscStubs.c, wintest/MsgStubs.c)

Der Interpretierer 110 erwartet, in einer Umgebung zu laufen, in der alle AOI-Systemaufrufe verfügbar sind. Um eine ausführbare Anweisung auch nur zu kompilieren und zu verbinden, müssen Fragmente für die AOI-Systemaufrufe geschaffen werden. Viele der Systemaufrufe haben keine Signifikanz, wenn das System unter Windows getestet wird, und so werden diese Aufrufe einfach als Leerfunktionen belassen (nur da für Verbindungszwecke). Implementationen der AOI-Systemaufrufe werden zur Zeitsteuerung (ScGtmSet, ScGtmRef) und für messsgAlc, ScMsgSnd, ScMsgRcv verwendet.

Der OOCT ist stark von den Systemaufrufen zur Nachrichtenweiterleitung für die Interprozeßkommunikation zwischen den Ausführungen und dem Kompilierer 104 abhängig. Unter Windows wird eine Scheinversion dieser AOI-Systemaufrufe verwendet, um zu ermöglichen, daß Anknüpfungen innerhalb derselben Aufgabe kommunizieren (siehe oben). Die Windows-Version der Nachrichtensystemaufrufe implementiert die vollständige Spezifikation der Systemaufrufe unter Verwendung von Verriegelungen und Nachrichtenwarteschlangen.

GETRENNTE ANKNÜPFUNGEN (THREADS) FÜR OMPILIERER/AUSFÜHRUNGEN

Um die Implementation und Diagnose unter Windows zu vereinfachen, wurden getrennte Anknüpfungen für den Kompilierer 104 und den Interpretierer 110 anstelle von getrennten Prozessen verwendet. Die Verwendung von Anknüpfungen vereinfacht die Nachrichtenweiterleitungsimplementation zwischen den beiden 'Aufgaben'. Ferner wird die Diagnose leichter, da sowohl ein einzelnes Diagnoseprogramm für beide Aufgaben (Interpretierer 110 und Kompilierer 104) verwendet werden kann, als auch das Diagnoseprogramm ausgebildet ist, auf mehrfachen Anknüpfungen zu arbeiten (uns ist kein Diagnoseprogramm bekannt, das Werkzeuge für eine Diagnose mehrfacher Prozesse aufweist).

VERGLEICHSEINHEIT

Der OOCT verwendet ein einzigartiges Testverfahren, das sich als äußerst wertvoll erwiesen hat. Da der OOCT kompilierte Code-Ergebnisse produzieren sollte, die genau gleich sind wie jene des Interpretierers 110, wurde ein Weg geschaffen, diese Ergebnisse direkt zu vergleichen. Unter der Windows-Testumgebung wurde eine Fähigkeit eingebaut, sowohl unter dem OOCT als auch dem Interpretierer 110 Programme laufen zu lassen, und automatisch Zwischenergebnisse zu vergleichen. Diese Vergleiche können willkürlich fein abgestuft werden, bis zu Prüfungen nach jeder Instruktion.

tion. Zusammen mit der Fähigkeit, das Verhalten von Programmen zu vergleichen, wurde ein automatischer Testgenerator geschrieben. Der Testgenerator schafft einen 'zufälligen' Code, der dann laufen gelassen und verglichen wird. Diese automatische Testgenerierung und dieser Vergleich sehen eine äußerst umfangreiche Programmfolge vor, um zu verifizieren, daß der OOCT korrekt arbeitet. Ferner wurde ein äußerst wertvoller Weg zur Lokalisation auftretender Störungen vorgesehen, da der automatische Vergleich auf den Platz zeigt, wo sich der kompilierte Code und der Interpretierer 110 erstmals unterscheiden.

Dieser Abschnitt beschreibt die Vergleichseinheit in zwei Stufen. Zuerst wird die Infrastruktur beschrieben, die zum Vergleichen der Ergebnisse des kompilierten Codes mit jenen des Interpretierers 110 verwendet wird. Als zweites wird die Generierung des Zufallscodes beim Testen beschrieben.

VERGLEICHSINFRASTRUKTUR

Die Vergleichsinfrastruktur basiert auf der Idee, zwei Versionen desselben K-Programms laufen zu lassen, wo der Maschinenzustand der simulierten K-Maschine (Register und Speicher) zu spezifischen Zeiten gezielt geprüft wird. Die Ergebnisse dieser Prüfpunkte werden dann verglichen, um zu bestimmen, ob die kompilierte Version und interpretierte Version dieselben Ergebnisse liefern.

Fig. 21 veranschaulicht insbesondere ein Beispiel des obigen Prozesses, der eine Vergleichsinfrastruktur gemäß einer Ausführungsform der vorliegenden Erfindung aufweist. In der Praxis wird der Vergleichstest als zwei Windows-Prozesse laufen gelassen. Der Stammprozeß läuft auf dem vollständigen OOCT-System mit einer Verzweigungsprotokollierung und Kompilierung. Der Nebenprozeß läuft nur als interpretierte Version von KOI. Beide Prozesse schreiben ihre Prüfpunktprotokolle in den Speicher (der Nebenprozeß schreibt in den gemeinsam genutzten Speicher), um ihren Effekt auf den simulierten K-Maschinenzustand aufzuzeichnen. Das Stammverfahren vergleicht die Daten in den Protokollen und berichtet über Diskrepanzen.

CODEGENERIERUNG

Die Generierung des Zufallscodes für die Vergleichstests wird von drei Einheiten durchgeführt. Zuerst sieht der K-Assembler einen Mechanismus zur Produktion des K-Maschinencodes unter Verwendung von C-Funktionsaufrufen vor. Als zweites werden Einheiten zur Schaffung verschiedener Arten von Basisblöcken von K-OP-Codes vorgesehen. Schließlich ermöglicht die Zufallssteuerflusseinheit die Generierung eines Codes mit vielen verschiedenen Steuerflußtypen.

K-Assembler (wintest/OOCT_Assemble.[h,c])

Der K-Assembler sieht einen geradlinigen Mechanismus zur Generierung des K-Codes aus dem Inneren eines C-Programms vor. Jeder K-OP-Code hat eine Funktion, die zur Assemblierung von Instruktionen spezifisch für diesen OP-Code verwendet wird. Die individuellen Instruktionen nehmen als Argumente einen Zeiger zum Speicher, wo der Code zu speichern ist, einen (möglicherweise leeren) Sprungmarkennamen, und ein Argument für jedes in der Instruktion verwendete Feld. Die Funktion kombiniert einfach die Felder in ihre korrekten Plätze und schreibt den Code in den Puffer. Da Verzweigungen zu einer Sprungmarke vor der Definition der Sprungmarke eintreten können, wird ein zweiter Durchlauf über den Code verwendet, um das Verzweigungsziel zu lösen.

Einheiten zur Schaffung des Zufalls-K-OP-Codes (wintest/GenArith.c, wintest/GenCassist.c, Wintest/GenMisc.C)

Zum Testen verschiedener Typen von Instruktionen wurden individuelle Einheiten geschaffen, welche Basisblöcke (geradliniger Code) generieren, die diese Typen von Instruktionen enthalten. Insbesondere werden Einheiten geschaffen, welche die arithmetischen und Verschiebungsoperationen, die C assist-Instruktionen und alle anderen vom OOCT implementierten Instruktionen generieren. Die Hauptschnittstelle zu den Einheiten erfolgt durch eine FillBasicBlock-Routine. Diese Routine nimmt als Argumente einen Speicherpuffer und eine Anzahl von Instruktionen, und schreibt die gegebene Anzahl von Instruktionen (zufällig gewählt) in den Puffer. Die FillBasicBlock-Routine wählt zufällig aus einem Array von Instruktionen generierenden Funktionen, um Instruktionen hinzuzufügen. Die Einheiten enthalten eine Instruktionen generierende Funktion für jeden K-OP-Code, der generiert werden kann. Diese Instruktionen generierende Funktion wählt geeignete Zufallswerte für die Argumente an den Assembler und assembliert die Instruktionen. Instruktionen werden nicht völlig zufällig generiert. Statt dessen werden sie mit bestimmten Einschränkungen generiert. Wenn beispielsweise ein Register zufällig als Ziel gewählt wird, werden niemals die Basisregister verwendet. Der Code ist auch auf die Verwendung einer Anzahl vordefinierter Speicherstellen beschränkt. In unseren Tests haben sich diese Beschränkungen nicht als sehr signifikant erwiesen. Wenn sie sich in der Zukunft als signifikant erweisen, ist es möglich, einige der Einschränkungen unter Verwendung eines komplexeren Prozesses zu reduzieren.

Die Verwendung von Zufallstests ist wichtig, da Testinteraktionen zwischen vielen verschiedenen Instruktionen getestet werden, was für einen Kompilierer 104 wie den OOCT besonders wichtig ist. Im OOCT kann sich der durch das Kompilieren einer Instruktion produzierte Code in Abhängigkeit von umgebenden Instruktionen wesentlich unterscheiden.

Fig. 22 veranschaulicht insbesondere ein Beispiel der Codegenerierung für dieselbe Instruktion mit verschiedenen umgebenden Instruktionen. Ferner testen Zufallstests viele Fälle, die Programmierer nicht testen würden.

Die Einheiten zur Schaffung von Zufalls-K-OP-Codes sind an sich für bestimmte Testtypen effektiv. Bei der Implementation eines neuen OP-Codes hat es sich beispielsweise als sehr effektives Verfahren erwiesen, eine einfache Schleife zu schaffen, die einen Basisblock von Instruktionen unter Verwendung dieses OP-Codes ausführt. Während die individuellen Einheiten effektiv sein können, ist zum vollständigen Testen bestimmter Aspekte des Kompilierers 104 ein kom-

plexerer Steuerfluß erforderlich.

Die Einheit zur Schaffung eines Zufallssteuerflusses (winest/Gdom control flow creation unit (GenControl)) wird zur Schaffung von Tests verwendet, die komplexere Typen von Steuerflüssen verwenden als geradlinige Codes. GenControl startet mit einem einzelnen Basisblock und führt eine bestimmte Anzahl von Transformationen durch (zufällig gewählt).

Die Transformationen, die derzeit durchgeführt werden, sind wie folgt:

Ein Basisblock kann in zwei Basisblöcke geteilt werden.

Ein Basisblock kann durch ein Rhombuszeichen ersetzt werden. Dies repräsentiert eine bedingte Verzweigung, wo sich die zwei Pfade wieder miteinander verbinden.

Ein Basisblock kann durch eine Schleife ersetzt werden.

Ein Basisblock kann durch drei Basisblöcke ersetzt werden, wobei ein Funktionsaufruf an den zweiten Basisblock erfolgt und zum dritten zurückspringt.

Nachdem die spezifizizierte Anzahl von Transformationen an den Basisblöcken durchgeführt wurde, existiert ein zufällig generierter Steuerflußgraph, der mit Instruktionen gefüllt werden muß. Dies besteht aus zwei Teilen. Zur Generierung des Codes für die Basisblöcke selbst werden die Einheiten zur Schaffung eines Zufalls-K-OP-Codes verwendet, die im vorhergehenden Abschnitt diskutiert wurden. Der zweite Teil ist, Instruktionen zur Durchführung der Verzweigungen und Schleifen einzufüllen. Schleifen verwenden eine vordefinierte Schablone, die eine Anzahl von Malen iteriert. Für bedingte Verzweigungen wird eine Zufallstestinstruktion verwendet.

KOMPIILERERCODEAUSZÜGE

Für Diagnosezwecke und für Optimierungszwecke werden zahlreiche Codeauszugsmechanismen im OOCT unter Windows verwendet. Es gibt zwei Hauptauszugsmechanismen. Erstens kann während der Kompilierung ein Auszug einer Codeliste vorgenommen werden, welche die K-OP-Codes enthält, die kompiliert werden, die IL, und (wenn er generiert wurde) den Zielcode. Der zweite Typ eines Auszugs ist ein Auszug des Zielcodes in eine Assemblierungsform, die für Testzwecke rückkompiliert und gegenverbunden werden kann.

Durch das Vornehmen eines Auszugs einer Kopie des IL-Codes nach bestimmten Stufen kann der Effekt eines gegebenen Kompilierer-104-Optimierungsdurchlaufs auf Korrektheit und Effektivität untersucht werden. Ferner kann durch die Untersuchung des produzierten Endcodes manuell untersucht werden, wie gut der Kompilierer 104 jeden K-OP-Code in die IL übersetzt, und die Qualität des für jede IL-Instruktion und jeden K-OP-Code produzierten Zielcodes. Diese Code-Auszüge werden unter Verwendung des COMBDUMP-Makros gesteuert, das zwischen Kompilierer-104-Durchläufen in OOCT_Optimize_IL_And_Gen_Code eingefügt wird (siehe compiler/ooct_trace.c). Dieses Makro ruft die OOCT_Combdump-Prozedur auf (siehe ooct_combdump.c), die über die K-OP-Codes und die IL-Instruktionen iteriert.

Aktuelle Profilierungswerkzeuge für Windows behandeln dynamisch generierte Codes nicht korrekt. Daher wird der zweite Typ eines Auszugs verwendet, so daß der dynamische Code von einem Lauf als statischer Code für einen anderen Lauf verwendet und korrekt profiliert werden kann. Dies wird in zwei Schritten erreicht. Im ersten Schritt wird das Programm mit der OC_DUMP-Flagge kompiliert (siehe compiler/ooct_dump.h), was bewirkt, daß jede K-OP-Code-Spur, die kompiliert ist, aufgezeichnet wird, und daß ein Auszug des Codes in eine Datei in einem rückkompilierbaren Format vorgenommen wird. Zweitens wird das Programm kompiliert und mit der OC_USEDUMP-Flagge laufen gelassen (siehe compiler/ooct_dump.h), welche die dynamische Kompilierung für den vorher kompilierten Code anstelle der Verwendung der statischen Version ausschaltet. Diese Version des Programms kann dann mit einem Profiler laufen gelassen werden, um eine Statistik über die Qualität des Codes aufzuzeichnen.

ZWEITE AUSFÜHRUNGSFORM DER VORLIEGENDEN ERFINDUNG

DYNAMISCHE OPTIMIERENDE OBJEKTCODE-ÜBERSETZUNG

KURZFASSUNG DER ZWEITEN AUSFÜHRUNGSFORM

Eine Architekturemulation ist die Imitation einer Computerarchitektur durch eine andere Computerarchitektur, so daß der Maschinencode für die Originalarchitektur ohne Modifikation laufen gelassen werden kann. Eine Objektcode-Übersetzung ist der Prozeß der Übersetzung eines Maschinencodes für eine Computerarchitektur in den Maschinencode für eine andere Computerarchitektur. Das beschriebene dynamische optimierende Objektcode-Übersetzungssystem verwendet Kompiliereroptimierungstechniken, um eine höhere Leistung zu erzielen als eine auf einer Schablone basierende Objektcode-Übersetzung für eine Architekturemulation.

BESCHREIBUNG VON FIGUREN DER ZWEITEN AUSFÜHRUNGSFORM

Fig. 23 veranschaulicht eine Systemkonfiguration, die für die dynamische optimierende Objektcode-Übersetzung gemäß der zweiten Ausführungsform der vorliegenden Erfindung verwendet wird. Fig. 23 ist eine schematische Darstellung einer dynamischen Übersetzung gleichzeitig mit der interpretierten Ausführung von Programmen. Jeder Interpretierer kann Übersetzungsanforderungen an den Kompilierer senden. Der Kompilierer macht dann einen übersetzten Code für die Interpretieraufgaben verfügbar. Auf einer Maschine mit mehrfachen Ausführungseinheiten können alle Prozesse gleichzeitig ausführen.

DETAILLIERTE BESCHREIBUNG DER ZWEITEN AUSFÜHRUNGSFORM

Das dynamische optimierende Objektcode-Übersetzungssystem nimmt eine dynamische Kompilierung eines Instruktionssatzes in einen anderen vor, um eine Leistungsverbesserung gegenüber einer auf einer Schablone basierenden Über-

setzung oder interpretierten Einheiten vorzusehen. Das dynamische optimierende Objektcode-Übersetzungssystem kombiniert eine beliebige Anzahl von Interpretierern, die eine Profilierung des laufenden Codes vornehmen, mit einem getrennten optimierenden Kompilierer. Der optimierende Kompilierer verwendet die Profilierungsinformationen aus dem laufenden Code, um häufig ausgeführte Teile des Codes zu bestimmen. Diese Teile werden dann kompiliert und den Interpretierern zur Verwendung geliefert. Die Gesamtstruktur des Systems ist in Fig. 23 gezeigt.

Die Durchführung bedeutender Optimierungen vom Kompilierer-Typ ist nur mit der Kenntnis des Instruktionsflußgraphen möglich. In einem traditionellen Kompilierer ist der Flußgraph gegeben und gut definiert, da die gesamte Routine einer vollständigen Sprachanalyse unterzogen wird, bevor die Optimierung beginnt. Für ein Architecturemulationsssystem ist der zu kompilierende Code nicht verfügbar, bevor er tatsächlich laufen gelassen wird. Ferner können Instruktionen und Daten allgemein nicht differenziert werden, ohne daß tatsächlich ein Programm laufen gelassen wird.

Daher muß zur Bestimmung des Flußgraphen das Programm laufen gelassen werden. Ein Interpretierer wird verwendet, um das Programm das erste Mal laufen zu lassen. Während der Interpretierer das Programm ausführt, informiert er den dynamischen Kompilierer jedesmal, wenn er eine Verzweigungsoperation durchführt. Diese Informationsprotokollierung identifiziert einige der Instruktionen und einige der Verbindungspunkte. Während das Programm läuft, werden die Informationen über den Flußgraphen vollständiger, obwohl sie nie ganz vollständig werden. Das System ist ausgebildet, mit Teilm Informationen über den Flußgraphen zu arbeiten: die Optimierung wird an potentiell unvollständigen Flußgraphen durchgeführt, und das System ist ausgebildet zu gestatten, daß ein optimierter Code ersetzt wird, während mehr Informationen verfügbar werden.

Die dynamische Kompilierung wählt, welche Teile des Texts zu optimieren sind, auf der Basis der vom Interpretierer gesammelten Profilierungsinformationen. Wenn die Anzahl von Malen, die irgendeine Verzweigung ausgeführt wird, eine Schwelle überschreitet, wird das Ziel dieser Verzweigung ein Startparameter zur Kompilierung. Der Startparameter ist ein Startpunkt für eine Sprachanalyse eines Teils der als Einheit zu kompilierenden Quelleninstruktionen. Diese Einheit wird als Segment bezeichnet.

Ein Segment enthält die Instruktionen, die aus der Optimierung der Quelleninstruktionen aus dem Startparameter resultieren. Es wird als Einheit installiert und deinstalliert. Wenn der Interpretierer den Kompilierer aufruft, um ihn über eine Verzweigung zu informieren, kann er wählen, die Steuerung in das Segment zu transferieren, wenn der Code für das Ziel existiert. Ähnlich kann das Segment einen Code für den Transfer der Steuerung zurück zum Interpretierer enthalten.

Ein Segment kann unvollständig sein, wobei es nur einen Subsatz der möglichen Flußpfade aus dem Quellenprogramm repräsentiert. Diese unvollständige Repräsentation beeinflusst jedoch nicht den korrekten Betrieb der Emulation. Wenn ein neuer, unvorhergesehener Flußpfad durch den Originalcode entsteht, dann springt der Steuerfluß zum Interpretierer zurück. Später kann dasselbe Segment ersetzt werden, um den neuen Steuerfluß zu berücksichtigen.

BESONDERE OBJEKTE DER ZWEITEN AUSFÜHRUNGSFORM

Die Erfindung ist die Verwendung der optimierten Objektcode-Übersetzung für eine verbesserte Leistung in Architecturemulationsystemen.

ZUSAMMENFASSUNG DER ZWEITEN AUSFÜHRUNGSFORM

Das beschriebene dynamische optimierende Objektcode-Übersetzungssystem verwendet Kompiliereroptimierungstechniken, um eine höhere Leistung zu erzielen als eine auf einer Schablone basierende Objektcode-Übersetzung für eine Architecturemulation. Die Erfindung ist die Verwendung der optimierten Objektcode-Übersetzung für eine verbesserte Leistung in Architecturemulationsystemen.

DRITTE AUSFÜHRUNGSFORM

GLEICHZEITIGE DYNAMISCHE ÜBERSETZUNG

KURZFASSUNG DER DRITTEN AUSFÜHRUNGSFORM

Die dynamische Übersetzung ist ein Akt der Übersetzung eines Computerprogramms in einer Maschinensprache in eine andere Maschinensprache, während das Programm läuft. Das beschriebene gleichzeitige dynamische Übersetzungssystem führt eine Übersetzung gleichzeitig mit interpretierter Programmausführung durch.

BESCHREIBUNG VON FIGUREN DER DRITTEN AUSFÜHRUNGSFORM

Fig. 24 veranschaulicht eine Systemkonfiguration, die für die gleichzeitige dynamische Übersetzung gemäß der dritten Ausführungsform der vorliegenden Erfindung verwendet wird. Fig. 24 ist eine schematische Darstellung einer dynamischen Übersetzung gleichzeitig mit interpretierter Ausführung von Programmen. Jede Interpretiereraufgabe kann Übersetzungsanforderungen an die Kompiliereraufgabe senden. Die Kompiliereraufgabe macht dann den übersetzten Code für die Interpretiereraufgaben verfügbar. Auf einer Maschine mit mehrfachen Ausführungseinheiten können alle Prozesse gleichzeitig ausgeführt werden.

Fig. 25 veranschaulicht den Unterschied zwischen der Kombination eines Interpretierers und Kompilierers, beispielsweise während der Ausführung als eine Aufgabe, und ihrer Trennung, beispielsweise in verschiedene Aufgaben, gemäß einer vierten Ausführungsform der vorliegenden Erfindung. Fig. 25 ist eine schematische Latenzdarstellung mit kombinierten und getrennten Interpretierer- und Kompiliereraufgaben.

Der Zweck der gleichzeitigen dynamischen Übersetzung ist, eine Leistungssteigerung gegenüber einem Interpretierer durch das Kompilieren eines Ausführungsprogramms in eine effizientere Form, während der Interpretierer noch läuft, vorzusehen. Zur Durchführung der dynamischen Übersetzung gleichzeitig mit der Ausführung eines Interpretierers läuft der Kompilierer als getrennte Aufgabe auf einem System mit mehrfachen Ausführungseinheiten. Die Kompiliereraufgabe ist ein Server, der Anforderungen zur Übersetzung einiger Instruktionen empfängt, und mit einem Stück übersetzten Code antwortet. Die Anordnung des Kompiliererservers als getrennte Aufgabe hat einige Vorteile. Erstens kann mehr als eine Interpretiereraufgabe an denselben Server Anforderungen stellen. Zweitens müssen die Interpretiereraufgaben nicht auf das Ergebnis einer Kompilierungsanforderung warten, bevor sie weitergehen. Drittens sind die Interpretierer und der Kompilierer gegen Fehler in anderen Aufgaben isoliert. Viertens können die Interpretierer und der Kompilierer unabhängig terminisiert werden, so daß die Arbeit gleichmäßiger über die Anzahl verfügbarer Prozessoren aufgeteilt wird. Jeder dieser Vorteile wird nachstehend detaillierter beschrieben.

Es gibt einige existierende dynamische Übersetzungssysteme, die keine getrennten Kompiliereraufgaben haben. Die virtuelle Java-Maschine von Sun Microsystems ist ein Beispiel [2]. Der Interpretierer in der virtuellen Maschine kann eine dynamische Übersetzungsanforderung ausgeben, indem er eine Prozedur aufruft. Der Interpretierer muß auf die Vollendung der Übersetzungsanforderung warten, bevor er die Ausführung des Programms fortsetzt. Ein weiteres Beispiel ist das dynamische OCT-Übersetzungssystem von Fujitsu, das eine Seite von Instruktionen zu einer Zeit übersetzt [1]. Im OCT-Übersetzungssystem muß der Interpretierer auf die Vollendung der Übersetzungsanforderung warten, bevor er die Ausführung fortsetzt.

Es sind auch Übersetzungsserver für die statische Übersetzung des Java-Quellencodes in den Java-Bytecode verfügbar [3]. Diese Server bieten die Vorteile einer getrennten Kompiliereraufgabe zur statischen Übersetzung, jedoch nicht zur dynamischen Übersetzung, da sie nicht operieren, während das Java-Programm läuft.

Der erste Vorteil der Anordnung der getrennten Kompiliereraufgabe ist, daß mehrfache Interpretiereraufgaben Übersetzungsanforderungen an denselben Server stellen können. Sie müssen den Kompilierercode nicht in ihrem ausführbaren Bild enthalten, wodurch es viel kleiner wird. Sie haben keine Cache-Konflikte zwischen Interpretiererinstruktionen und Kompiliererinstruktionen oder zwischen Interpretiererdaten und Kompiliererdaten. Da auf nahezu allen modernen Prozessoren eine effiziente Cache-Nutzung wichtig ist, stellt dies einen signifikanten Vorteil dar.

Der zweite Vorteil einer getrennten Kompiliereraufgabe ist, daß die Interpretierer die Latenz des Kompilierers nicht sehen. Fig. 25 veranschaulicht die Differenz der Latenz. Mit der kombinierten Interpretierer- und Kompiliereraufgabe führt der Interpretierer keine Instruktionen aus, bis der Kompilierer die Übersetzung der Instruktionen beendet hat. Mit den getrennten Aufgaben nimmt der Interpretierer sofort die Ausführung von Instruktionen wieder auf, während der Kompilierer arbeitet. Die gesamte von den getrennten Aufgaben verrichtete Arbeit ist größer, da sie Übersetzungsanforderungen senden und empfangen müssen, die geringere Latenz bedeutet jedoch, daß Benutzer des Systems keine Pausen einhalten, während der Kompilierer arbeitet. Die Interpretiereraufgabe kann auch auf externe Ereignisse, wie Unterbrechungen, reagieren, während der Kompilierer arbeitet, was in der Anordnung der kombinierten Aufgaben nicht möglich sein kann. In der Praxis setzt die Tatsache, daß der Interpretierer die Latenz des Kompilierers in der kombinierten Anordnung sieht, eine Grenze für die Komplexität des Kompilierers und die Qualität des übersetzten Codes. Beispielsweise sollten Java Just-In-Time-Kompilierer schnell genug ausführen, so daß ein mit dem Java-System interagierender Benutzer keine Pause sieht, was einige komplexe Optimierungen ausschließt. Ähnlich führt das OCT-System nur eine Optimierung innerhalb einer einzigen übersetzten Instruktion durch, um die Kompilierungszeit zu reduzieren. Die Anordnung der getrennten Kompiliereraufgabe ermöglicht eine Optimierung quer über mehrfache Instruktionen.

Der dritte Vorteil der getrennten Kompiliereraufgabe ist, daß Fehler in den Interpretiereraufgaben und der Kompiliereraufgabe voneinander isoliert werden. Dies bedeutet, daß, wenn die Kompiliereraufgabe eine Adressenausnahme oder Ausnahmebedingung erhält, die Interpretiereraufgabe nicht beeinträchtigt wird. Der Kompilierer setzt sich selbst nach einem Fehler zurück und setzt die Arbeit an der nächsten Anforderung fort. Da die Interpretiereraufgaben nicht warten, bis der Kompilierer eine Übersetzungsanforderung beendet hat, merken sie nicht, wenn der Kompilierer einen Fehler erhält.

Der vierte Vorteil der getrennten Kompiliereraufgabe ist, daß sie die Belastung der Kompilierer- und Interpretiereraufgaben ausgleichen kann. Im dynamischen Übersetzungssystem gibt es Zeiten, wenn die Interpretiereraufgaben stark belegt sind und alle CPUs des Computers benötigen, und es gibt Zeiten, wenn die Interpretiereraufgaben untätig sind und die CPUs nicht genutzt werden. In der kombinierten Interpretierer- und Kompiliereranordnung wird die meiste Kompilierungsarbeit verrichtet, wenn die Interpretierer belegt sind, da der Kompilierer nur aufgerufen wird, wenn der Interpretierer läuft. Dies nützt die untätigen CPU-Zyklen nicht aus. In der Anordnung der getrennten Kompiliereraufgabe setzt der Kompilierer die Arbeit fort, wenn die Interpretierer untätig sind. Sie produziert einen übersetzten Code, den die Interpretierer wahrscheinlich in der Zukunft verwenden werden.

BESONDERE OBJEKTE DER DRITTEN AUSFÜHRUNGSFORM

Die dritte Ausführungsform der vorliegenden Erfindung ist auf die Verwendung einer dynamischen Übersetzung gleichzeitig mit mehrfachen Interpretierern, die auf einem System ausführen, mit mehrfachen physischen Ausführungseinheiten gerichtet, wobei eine kleinere ausführbare Bildgröße, eine reduzierte Cache-Konkurrenz, eine geringere Interpretiererausführungslatenz, eine Fehlerisolierung und ein besserer Belastungsausgleich vorgesehen werden.

ZUSAMMENFASSUNG DER DRITTEN AUSFÜHRUNGSFORM

Das beschriebene dynamische Übersetzungssystem führt eine Übersetzung gleichzeitig mit interpretierter Programmausführung durch. Das System verwendet einen getrennten Kompilierer, so daß er die Leistung der Interpretiererauf-

gaben nicht signifikant beeinträchtigt. Die Erfindung ist die Verwendung einer dynamischen Übersetzung gleichzeitig mit mehrfachen Interpretierern, die auf einem System ausführen, mit mehrfachen physischen Ausführungseinheiten, wobei eine kleinere ausführbare Bildgröße, eine reduzierte Cache-Konkurrenz, eine geringere Interpretiererausführungslatenz, eine Fehlerisolierung und ein besserer Belastungsausgleich vorgesehen werden.

VIERTE AUSFÜHRUNGSFORM DER VORLIEGENDEN ERFINDUNG

EMULATION WÄHREND DER DYNAMISCHEN ÜBERSETZUNG, UM DIE BELASTUNG DER PROFILIERUNG AUF DEN EMULATOR ZU REDUZIEREN

KURZFASSUNG DER VIERTEN AUSFÜHRUNGSFORM

Eine Architektur emulation ist die exakte Imitation einer Computerarchitektur durch eine andere Computerarchitektur, so daß der Maschinencode für die Originalarchitektur ohne Modifikation laufen gelassen werden kann. Eine Objektcode-Übersetzung ist der Prozeß der Übersetzung eines Maschinencodes für eine Computerarchitektur in den Maschinencode für eine andere Computerarchitektur. Das beschriebene dynamische optimierende Objektcode-Übersetzungssystem verwendet Kompiliereroptimierungstechniken, um eine höhere Leistung zu erzielen als eine auf einer Schablone basierende Objektcode-Übersetzung für eine Architektur emulation. Es ist jedoch eine Profilierung erforderlich, um die dynamische optimierende Objektcode-Übersetzung zu realisieren. Die Beschreibung erläutert ein Verfahren zur Reduktion der Belastung der Profilierung.

BESCHREIBUNG VON FIGUREN DER VIERTEN AUSFÜHRUNGSFORM

Fig. 26 veranschaulicht eine Übersetzungstabelle, die verwendet wird, um aufzuzeichnen, welche Instruktionen übersetzbar sind und welche nicht, gemäß einer vierten Ausführungsform der vorliegenden Erfindung. Fig. 26 ist eine Übersetzungstabelle, die zeigt, welche Programme übersetzbar sind und welche nicht. In diesem Fall werden Programme in Einheiten von 1 Bytes gemessen. Der Emulator prüft, welchem Eintrag ein Verzweigungsnachfolger entspricht, wodurch bestimmt wird, ob er zu einem übersetzbaren Programm springt oder nicht.

Fig. 27 veranschaulicht, wie das Verfahren die Belastung der Profilierung auf den Emulator gemäß einer vierten Ausführungsform der vorliegenden Erfindung reduziert. Fig. 27 ist ein Flußdiagramm, das zeigt, wie der Emulator die Protokollierung für übersetzbare Programme einschaltet, und wie er sie für nicht-übersetzbare Programme ausschaltet. Die Trigger *1- und Trigger *2-Instruktionen sollten beide protokolliert werden, die Trigger *1-Instruktion kann jedoch nicht zwischen einem übersetzbaren Programm und einem nichtübersetzbaren Programm springen. Nur die Trigger *2-Instruktion kann zwischen ihnen springen. Die Protokollflagge, die sich merkt, ob der Emulator in einem übersetzbaren oder nicht-übersetzbaren läuft. Daher muß der Emulator bei Trigger *1-Instruktionen nicht die Übersetzungstabelle prüfen oder die Protokollflagge ändern. Er prüft nur, ob die Verzweigungsnachfolgerinstruktion bereits kompiliert wurde, und springt sofort zum kompilierten Code. Da Trigger *1-Instruktionen die am häufigsten verwendeten Trigger-Instruktionen repräsentieren, kann dieser Algorithmus die Belastung der Profilierung auf die Emulation reduzieren.

DETAILLIERTE BESCHREIBUNG DER VIERTEN AUSFÜHRUNGSFORM

Die dynamische optimierende Objektcode-Übersetzung realisiert eine hohe Leistung durch die Produktion schnellerer Instruktionen, sie führt jedoch zu Kosten hinsichtlich Speicher und Zeit. Daher werden in einer Architektur emulation sowohl die dynamische optimierende Objektcode-Übersetzung als auch die Emulation gemeinsam verwendet. Die Übersetzung wird für das Hauptprogramm verwendet, das häufig läuft und eine hohe Leistung benötigt. Und der Emulator arbeitet für kleinere Programme und auch zur Profilierung des Hauptprogramms, bis der Übersetzer die Kompilierung vollendet. Ein Profil wird vom Übersetzer verwendet, um das Programm zu kompilieren und zu optimieren.

Instruktionen, die von einem nicht-übersetzten Code zu einem übersetzten Code springen können, werden als Triggerinstruktionen bezeichnet. Wenn eine Triggerinstruktion von einem Nebenprogramm zu einem Hauptprogramm oder von einem Hauptprogramm zu einem Nebenprogramm springen kann, wird sie Trigger *2-Instruktion genannt. Wenn sie nur innerhalb eines Nebenprogramms oder eines Hauptprogramms springen kann, wird sie Trigger *1-Instruktion genannt. Da der Übersetzer nicht auf den Nebenprogrammen arbeitet, ist es nicht notwendig, die Trigger *1-Instruktionen in einem Nebenprogramm zu profilieren. Es ist notwendig, Trigger *1-Instruktionen in einem Hauptprogramm zu profilieren, da ein Teil des Programms übersetzt sein kann, während ein anderer Teil noch nicht übersetzt ist. Es ist notwendig, Trigger *2-Instruktionen sowohl in Neben- als auch Hauptprogrammen zu profilieren, da sie in ein anderes Hauptprogramm springen könnten.

Die Emulation nimmt drei Prüfungen nach der Ausführung einer Trigger *2-Instruktion vor (siehe Fig. 27). Zuerst prüft sie, ob der Übersetzer ein ist. Wenn er ein ist, prüft sie, ob der Nachfolger der Trigger *2-Instruktion übersetzbar ist oder nicht. Wenn er übersetzbar ist, dann setzt die Emulation die Protokollierflagge auf wahr und prüft, ob der Nachfolger übersetzt wurde, wobei sie zur übersetzten Version springt, wenn diese existiert.

Die Emulation nimmt nur zwei Prüfungen nach der Ausführung einer Trigger *1-Instruktion vor (siehe Fig. 27). Zuerst prüft sie, ob die Protokollierflagge ein oder aus ist. Wenn sie Flagge aus ist, dann ist diese Instruktion in einem Nebenprogramm, und sie muß nicht profiliert werden. Wenn die Flagge ein ist, dann prüft die Emulation, ob ihr Nachfolger übersetzt wurde oder nicht.

Haupt- und Nebenprogramme werden durch ihre Speicheradressen unterschieden (siehe Fig. 27). Der Emulator verwendet eine Übersetzungstabelle, um die Beziehung von übersetzbaren und nicht-übersetzbaren Programmadressen aufzuzeichnen. Für Trigger *1-Instruktionen, die sich niemals zwischen übersetzbaren Programmen und nicht-übersetzbaren Programmen bewegen, muß der Emulator nicht auf die Übersetzungstabelle zugreifen, da die Protokollierflagge

diese Informationen bereits enthält.

Durch die Trennung des Verhaltens des Emulators für Trigger *1- und Trigger *2-Instruktionen in zwei Verfahren wird die Belastung der Profilierung auf die Emulation reduziert.

BESONDERE OBJEKTE DER VIERTEN AUSFÜHRUNGSFORM

Die vierte Ausführungsform der vorliegenden Erfindung ist auf ein Verfahren zur Reduktion der Belastung der Profilierung auf den Emulator gerichtet, indem ein Code nach Triggerinstruktionen plazierte wird, die in und aus übersetzbaren Instruktionen springen können, der prüft, ob der Verzweigungsnachfolger übersetzbar ist oder nicht, und indem ein Code nach allen anderen Triggern plazierte wird, der nur eine Flagge prüft, um zu sehen, ob sie übersetzbar ist oder nicht.

ZUSAMMENFASSUNG DER VIERTEN AUSFÜHRUNGSFORM

Es ist effektiv, die dynamische Objektcode-Übersetzung zusammen mit der Emulation zu verwenden, aber die Kosten von Profilierungsinstruktionen zur Führung des Übersetzers sind eine Belastung für die Emulation. Durch die Unterscheidung zwischen verschiedenen Typen profilierter Instruktionen ist es möglich, diese Belastung zu reduzieren. Die Erfindung ist ein Verfahren zur Reduktion der Belastung der Profilierung auf den Emulator, indem ein Code nach Trigger-Instruktionen plazierte wird, die in und aus übersetzbaren Instruktionen springen können, der prüft, ob der Verzweigungsnachfolger übersetzbar ist oder nicht, und indem ein Code nach allen anderen Triggern plazierte wird, der nur eine Flagge prüft, um zu sehen, ob sie übersetzbar ist oder nicht.

FÜNFTE AUSFÜHRUNGSFORM DER ERFINDUNG

SOFTWARE-RÜCKKOPPLUNG FÜR DIE DYNAMISCHE ÜBERSETZUNG

KURZFASSUNG DER FÜNFTEN AUSFÜHRUNGSFORM

Die dynamische Übersetzung ist ein Akt der Übersetzung eines Computerprogramms in einer Maschinensprache in eine andere Maschinensprache, während das Programm läuft. In einigen dynamischen Übersetzungssystemen ist die Aufgabe, die das Programm laufen läßt, Interpretierer genannt, von der Aufgabe, die das Programm übersetzt, Kompilierer genannt, getrennt. Die Rate, bei welcher der Interpretierer Anforderungen an den Kompilierer sendet, sollte mit der Rate übereinstimmen, bei welcher der Kompilierer die Anforderungen vollendet. Auch sollte die Rate, bei welcher der Interpretierer Anforderungen sendet, nicht auf Null fallen. Die Software-Rückkopplung sieht einen Weg zum Ausgleichen der beiden Raten vor.

BESCHREIBUNG VON FIGUREN DER FÜNFTEN AUSFÜHRUNGSFORM

Fig. 28 veranschaulicht eine allgemeine Strukturdarstellung eines dynamischen Übersetzungssystems mit getrenntem Interpretierer und Kompilierer gemäß einer fünften Ausführungsform der vorliegenden Erfindung. Fig. 28 ist eine Strukturdarstellung eines dynamischen Übersetzungssystems. Der Interpretierer sendet Übersetzungsanforderungen an den Kompilierer. Der Kompilierer sendet einen übersetzten Code als Antwort zurück. Die Raten der Anforderungen und Antworten sollten gleich sein, damit das System am effizientesten läuft.

Fig. 29 veranschaulicht Komponenten eines Software-Rückkopplungsmechanismus gemäß einer fünften Ausführungsform der vorliegenden Erfindung. Fig. 29 ist eine Darstellung, die Komponenten eines Software-Rückkopplungssystems veranschaulicht. Die Vergleichsprozeder subtrahiert die Anzahl der Vollendungen von der Anzahl der Anforderungen. Die Anforderungsratenprozedur stellt die Rate auf der Basis dieser Differenz ein. Die Anforderungssende-prozedur sendet Anforderungen in Abhängigkeit von der aktuellen Rate.

DETAILLIERTE BESCHREIBUNG DER FÜNFTEN AUSFÜHRUNGSFORM

In einem dynamischen Übersetzungssystem sendet die Interpretiereraufgabe Anforderungen an die Kompiliereraufgabe. Die Anforderungen enthält Informationen, um dem Kompilierer mitzuteilen, welcher Abschnitt des Programms zu übersetzen ist. Der Kompilierer übersetzt den Abschnitt und antwortet mit einem übersetzten Code. Das Problem der Entscheidung, wann eine Anforderung zu senden ist, ist ein Beispiel eines Terminisierungsproblems. Die Rate, bei welcher die Interpretiereraufgabe Anforderungen stellt, sollte mit der Rate übereinstimmen, bei welcher der Kompilierer Anforderungen beendet. So wird der Kompilierer nicht untätig oder mit Anforderungen überladen.

Die Software-Rückkopplung ist ein Verfahren zum Ausgleichen der Raten von zwei Ereignissätzen [1]. Im dynamischen Übersetzungssystem ändert sie die Rate von Übersetzungsanforderungen, um gleich der Rate vollendeter Übersetzungen zu sein. Wie in Fig. 29 gezeigt hat das Software-Rückkopplungssystem drei Hauptteile. Der erste ist eine Prozedur zum Vergleichen der Anzahl von Übersetzungsanforderungen und der Anzahl vollendeter Übersetzungen. Der zweite ist eine Prozedur, welche die Rate von Übersetzungsanforderungen auf der Basis des Ergebnisses des Vergleichs ändert. Der dritte Teil ist eine Prozedur, um die Übersetzungsanforderungen zu stellen, die vom Ausgang der zweiten Prozedur abhängig sind.

Im dynamischen Übersetzungssystem zählt die Interpretiereraufgabe, wie oft eine Verzweigungsinstruktion zu einer bestimmten Zieladresse springt. Wenn diese Zählung eine Schwelle überschreitet, sendet der Interpretierer eine Übersetzungsanforderung, welche die Zieladresse enthält. Der Schwellenwert ist der kritische Parameter, der vom Software-Rückkopplungsmechanismus eingestellt wird. Wenn die Schwelle niedriger ist als die meisten der Ausführungszählungen, ist die Rate von Übersetzungsanforderungen hoch. Wenn die Schwelle höher ist als die meisten der Ausführungs-

zählungen, ist die Rate von Anforderungen niedrig. Da die typische Größe einer Anforderungszählung mit dem Programm, das interpretiert wird, variiert, ist die Software-Rückkopplung ein idealer Weg, die Schwelle einzustellen, da sie sich an das Verhalten des Interpretierers automatisch anpaßt.

Im dynamischen Übersetzungssystem ist die Vergleichsprozedur des Software-Rückkopplungssystems sehr einfach. Sie berechnet nur die Differenz zwischen der Anzahl von an den Kompilierer gesendeten Übersetzungsanforderungen und die Anzahl vollendeter Übersetzungen.

Die Anforderungsratenprozedur ändert den Schwellenwert auf der Basis der von der Vergleichsprozedur berechneten Differenz. Wenn die Differenz Null ist, dann ist die Schwelle zu hoch und hindert den Interpretierer daran, Übersetzungsanforderungen zu senden. In diesem Fall subtrahiert die Anforderungsratenprozedur eine Konstante von der Schwelle. Wenn die Differenz ihr maximal möglicher Wert ist, dann ist die Schwelle zu niedrig, und der Interpretierer sendet zu viele Übersetzungsanforderungen. In diesem Fall addiert die Anforderungsratenprozedur eine Konstante zur Schwelle.

Die Anforderungssendeprozedur wird aufgerufen, wenn der Interpretierer eine Verzweigungsanweisung ausführt. Wenn die Verzweigungsanweisung mehrere Male zur gleichen Zieladresse gesprungen ist als die Schwelle, sendet der Interpretierer eine Übersetzungsanforderung, welche die Zieladresse enthält.

BESONDERE OBJEKTE DER FÜNFTEN AUSFÜHRUNGSFORM

Die Erfindung ist die Verwendung eines Software-Rückkopplungsmechanismus in einem dynamischen Übersetzungssystem mit getrennten Interpretierer- und Kompiliereraufgaben, um die Rate von vom Interpretierer gesendeten Übersetzungsanforderungen und die Rate von vom Kompilierer vollendeten Übersetzungen auszugleichen, ohne zu ermöglichen, daß der Kompilierer untätig wird.

Die Verwendung einer minimalen Schwelle, um dem Kompilierer zu ermöglichen, den Betrieb einzustellen.

ZUSAMMENFASSUNG DER FÜNFTEN AUSFÜHRUNGSFORM

In einem dynamischen Übersetzungssystem mit getrennten Interpretierer- und Kompiliereraufgaben sollte die Rate, bei welcher der Interpretierer Anforderungen an den Kompilierer sendet, mit der Rate übereinstimmen, bei welcher der Kompilierer die Anforderungen vollendet. Auch sollte die Rate, bei welcher der Interpretierer Anforderungen sendet, nicht auf Null fallen. Die Erfindung ist die Verwendung eines Software-Rückkopplungssystems in einem dynamischen Übersetzungssystem mit getrennten Interpretierer- und Kompiliereraufgaben, um die Rate von vom Interpretierer gesendeten Übersetzungsanforderungen und die Rate von vom Kompilierer vollendeten Übersetzungen auszugleichen, ohne zu ermöglichen, daß der Kompilierer untätig wird.

SECHSTE AUSFÜHRUNGSFORM DER ERFINDUNG

EINREIHEN VON ANFORDERUNGEN IN EINE WARTESCHLANGE FÜR DIE DYNAMISCHE ÜBERSETZUNG

KURZFASSUNG DER SECHSTEN AUSFÜHRUNGSFORM

Die dynamische Übersetzung ist ein Akt der Übersetzung eines Computerprogramms in einer Maschinensprache in eine andere Maschinensprache, während das Programm läuft. Für jedes Teilstück des Programms, das übersetzt wird, stellt das System eine Anforderung an den dynamischen Übersetzer. Anforderungen, die gestellt werden, während der dynamische Übersetzer belegt ist, werden in eine Warteschlange eingereiht und ausgegeben, wenn der Übersetzer untätig wird. Die Implementation des Einreihens in eine Warteschlange kombiniert Systemaufrufe und gemeinsam genutzte Speicherkommunikation, zum Reduzieren.

BESCHREIBUNG VON FIGUREN DER SECHSTEN AUSFÜHRUNGSFORM

Fig. 30 veranschaulicht, wie eine Warteschlange verwendet wird, um Übersetzungsanforderungen zu halten, während die Übersetzungsaufgabe belegt ist, gemäß einer sechsten Ausführungsform der vorliegenden Erfindung.

Fig. 31 veranschaulicht, wie die OOCT-Anforderungswarteschlange kostengünstige gemeinsam genutzte Speicheranforderungen mit Systemaufrufanforderungen gemäß einer sechsten Ausführungsform der vorliegenden Erfindung kombiniert.

DETAILLIERTE BESCHREIBUNG DER SECHSTEN AUSFÜHRUNGSFORM

Die Grundfunktion der Anforderungswarteschlange ist, sich Anforderungen zu merken, die gestellt werden, während der dynamische Übersetzer belegt ist, wie in Fig. 30 gezeigt. In jedem dynamischen Übersetzungssystem gibt es eine Obergrenze der Anzahl von Übersetzungen, die gleichzeitig durchgeführt werden können. Typischerweise ist die Grenze nur eine Übersetzung zu einer Zeit. Es besteht jedoch keine Grenze für die Gesamtanzahl gestellter Anforderungen oder die Rate, bei der sie gestellt werden. Daher ist es sehr wahrscheinlich, daß eine Übersetzungsanforderung eintreten wird, während der Übersetzer bereits belegt ist. Mit einer Anforderungswarteschlange wird die Übersetzungsanforderung in eine Warteschlange eingereiht und muß nicht wiederholt werden. Wenn der Übersetzer die Anforderung aus der Warteschlange nimmt, wird er die Übersetzung durchführen.

Im OOCT hat das dynamische Übersetzungssystem mehrfache Aufgaben, wobei eine die dynamische Übersetzungsaufgabe ist, die Anforderungen behandelt, und andere die Ausführungsaufgaben sind, die Übersetzungsanforderungen stellen. Die OOCT-Implementation des Einreihens in eine Warteschlange verbessert eine naive Warteschlange unter Verwendung von weniger kostspieligem, gemeinsam genutztem Speicher zusammen mit Systemaufrufnachrichten, um die

Anforderungswarteschlange zu bilden, wie in Fig. 31 gezeigt. Systemaufrufe aller Art sind ausreichend, um Startparameter von den Ausführungsaufgaben zur Übersetzungsaufgabe zu kommunizieren, und um es der Übersetzungsaufgabe zu ermöglichen, untätig zu werden, oder zu blockieren, wenn es keine ausstehenden Anforderungen gibt. Systemaufrufe sind jedoch kostspielige Operationen. Der gemeinsam genutzte Speicher kann verwendet werden, um die Anforderungsnachrichten von den Ausführungsaufgaben zur Übersetzungsaufgabe zu kommunizieren, die Übersetzungsaufgabe kann jedoch diese Nachrichten nicht blockieren, so daß sie kontinuierlich laufen müßte, um Nachrichten von einem einfachen gemeinsam genutzten Speicher zu empfangen.

Die OOC-Implementation verwendet die besten Merkmale jedes Mechanismus, Systemaufruf und gemeinsam genutzter Speicher. Sie ermöglicht es der Übersetzungsaufgabe zu blockieren, während sie auf eine Systemaufrufnachricht wartet, kommuniziert jedoch Anforderungen durch den gemeinsam genutzten Speicher, wenn die Übersetzungsaufgabe bereits arbeitet.

Wie in Fig. 31 gezeigt, verwendet die OOC-Anforderungswarteschlange zwei Arten von Nachrichten zwischen den Ausführungs- und Übersetzungsaufgaben, plus einen gemeinsam genutzten Speicherpuffer, auf den von beiden Aufgaben zugegriffen wird. Die erste Nachricht geht von der Übersetzungsaufgabe zur Ausführungsaufgabe. Sie teilt der Ausführungsaufgabe mit, einen Systemaufruf zum Senden der nächsten Anforderung zu verwenden. Diese Nachricht informiert die Ausführungsaufgabe, daß die Übersetzungsaufgabe den gemeinsam genutzten Speicherpuffer entleert hat und nun blockieren wird. Dann sendet die Ausführungsaufgabe eine Anforderung mit einem Systemaufruf. Die Übersetzungsaufgabe empfängt die Nachricht und beginnt eine Übersetzung. Nach dem Senden einer Anforderung mit einem Systemaufruf weiß die Ausführungsaufgabe, daß die Übersetzungsaufgabe belegt ist, daher sendet sie weitere Anforderungen direkt an den gemeinsam genutzten Speicherpuffer. Dies ist viel weniger kostspielig als die Verwendung eines weiteren Systemaufrufs. Wenn die Übersetzungsaufgabe eine Anforderung beendet, sieht sie im gemeinsam genutzten Speicherpuffer nach. Wenn sich in dem Puffer eine Anforderung befindet, wird sie entfernt und übersetzt. Wenn der gemeinsam genutzte Speicherpuffer leer ist, teilt die Übersetzungsaufgabe wieder der Ausführungsaufgabe mit, einen Systemaufruf zu verwenden.

Die Vorteile der OOC-Anforderungswarteschlange sind, daß die Ausführungsaufgaben einen gemeinsam genutzten Speicher verwenden können, wenn sie Anforderungen bei einer hohen Rate senden, und die Übersetzungsaufgabe blockieren kann, wenn Anforderungen bei einer langsamen Rate ankommen.

BESONDERE OBJEKTE DER SECHSTEN AUSFÜHRUNGSFORM

Dieser Anspruch ist eine Übersetzung des Fujitsu-Patents auf japanisch, wobei ein Absatz hinzugefügt ist.

Die Erfindung ist ein Verfahren zum Fortsetzen einer Interpretation, während die Übersetzung häufig verzweigter Instruktionen gestartet wird, indem eine Nachricht an die Übersetzungsaufgabe gesendet wird, und zum Einreihen von Nachrichten an die Übersetzungsaufgabe in eine Warteschlange, wenn eine Übersetzung bereits begonnen hat, und eine Leistungsverbesserung gegenüber der Verwendung von sowohl Systemaufruf- als auch gemeinsam genutzten Speichermechanismen, um die Übersetzungsanforderungsnachrichten zu senden.

ZUSAMMENFASSUNG DER SECHSTEN AUSFÜHRUNGSFORM

Die beschriebene Übersetzungsanforderungswarteschlange ist ein Mechanismus zum Sammeln von Übersetzungsanforderungen, während eine andere Übersetzung ausführt. Sie ermöglicht, daß die Ausführungsaufgaben unmittelbar nach dem Senden einer Anforderung weiterlaufen. Durch die Verwendung sowohl von einem gemeinsam genutzten Speicher als auch Systemaufrufen miteinander ist es möglich, die Effizienz der Übersetzungswarteschlange zu verbessern. Die Erfindung ist ein Verfahren zum Fortsetzen einer Interpretation, während die Übersetzung häufig verzweigter Instruktionen gestartet wird, indem eine Nachricht an die Übersetzungsaufgabe gesendet wird, und zum Einreihen von Nachrichten an die Übersetzungsaufgabe in eine Warteschlange, wenn eine Übersetzung bereits begonnen hat, und eine Leistungsverbesserung gegenüber der Verwendung von sowohl Systemaufruf- als auch gemeinsam genutzten Speichermechanismen, um die Übersetzungsanforderungsnachrichten zu senden.

SIEBENTE AUSFÜHRUNGSFORM DER VORLIEGENDEN ERFINDUNG

SEITENFEHLERBEHEBUNG FÜR DIE DYNAMISCHE ÜBERSETZUNG

KURZFASSUNG DER SIEBENTEN AUSFÜHRUNGSFORM

Die dynamische Übersetzung ist ein Akt der Übersetzung eines Computerprogramms in einer Maschinensprache in eine andere Maschinensprache, während das Programm läuft. Der dynamische Übersetzer muß die Quellenmaschineninstruktionen lesen, bevor er sie in Zielmaschineninstruktionen übersetzt. Während die Quelleninstruktionen gelesen werden, kann der Übersetzer einen Seitenfehler verursachen, indem er aus einem Speicher liest, der ausgelagert ist, es ist jedoch ineffizient, den Speicher einzulagern. Der beschriebene Übersetzer behebt Seitenfehler, ohne die ausgelagerten Daten zu lesen, und setzt die Übersetzung fort.

BESCHREIBUNG VON FIGUREN DER SIEBENTEN AUSFÜHRUNGSFORM

Fig. 32 zeigt, wie ein dynamischer Übersetzer wahrscheinlich Seitenfehler verursachen kann, die während der normalen Ausführung der Quelleninstruktionen nicht eintreten würden, gemäß einer siebenten Ausführungsform der vorliegenden Erfindung.

Fig. 33 zeigt den Algorithmus zur Behebung von Seitenfehlern während der Übersetzung und zum Fortsetzen der

Übersetzung gemäß einer siebenten Ausführungsform der vorliegenden Erfindung.

DETAILLIERTE BESCHREIBUNG DER SIEBENTEN AUSFÜHRUNGSFORM

Ein dynamischer Übersetzer wird sehr wahrscheinlich auf Seiten zugreifen, die schlechte Kandidaten für das Kopieren in einen physischen Speicher sind, da er alle der möglichen Nachfolger einer Instruktion liest und nicht nur die Nachfolger, die tatsächlich ausgeführt werden. Beispielsweise, wie in Fig. 32 gezeigt, haben bedingte Verzweigungsinstruktionen zwei Nachfolger, den Fehlschlag-Nachfolger und den Verzweigung-ingeschlagen-Nachfolger. Wenn eine CPU eine bedingte Verzweigungsinstruktion ausführt, wenn die Verzweigung nicht eingeschlagen wird, dann wird die Verzweigung-ingeschlagen-Nachfolgerinstruktion niemals geladen. Daher wird sie keinen Seitenfehler verursachen. Wenn der dynamische Übersetzer die Verzweigungsinstruktion liest, versucht er, sowohl den Fehlschlag- als auch den Verzweigung-ingeschlagen-Nachfolger zu lesen, ohne zu wissen, welcher tatsächlich ausgeführt wird. Er könnte einen Seitenfehler verursachen, um die Verzweigungsnachfolgerinstruktion zu lesen, auch wenn sie niemals ausgeführt wird.

Das normale Verfahren zur Behandlung von Seitenfehlern ist, eine Seite in den angeforderten Speicher einzulagern, und den Speicherzugriff über Software vorzunehmen, und dann der Ausführung zu gestatten, nach der fehlerhaften Instruktion fortzusetzen. Dieses Verfahren hat zwei Nachteile. Erstens braucht es Zeit, eine Seite vom physischen Speicher zum Sicherungsspeicher zu bewegen, und eine andere vom Sicherungsspeicher zum physischen Speicher zu bewegen, und dann den Speicherzugriff durchzuführen. Zweitens wird der Satz von Speicherseiten, die darin eingelagert sind, geändert. Auf die Seite, die in den physischen Speicher kopiert wird, kann selten zugegriffen werden, bevor sie wieder ausgelagert wird, was bedeuten würde, daß es eine schlechte Idee war, sie in den physischen Speicher zu kopieren.

Da der dynamische Übersetzer häufigere Seitenfehler verursachen kann, ist es vorteilhaft, die Kosten dieser Seitenfehler zu reduzieren. Der dynamische Übersetzer minimiert die Kosten zusätzlicher Seitenfehler, indem er eine neue Seite nicht in den physischen Speicher kopiert, und eine Seite, die bereits im physischen Speicher ist, nicht entfernt. Dies spart die Kopierzeit und stellt auch sicher, daß eine Seite, auf die selten bezuggenommen wird, nicht einkopiert wird. Anstelle des Kopierens der Seite unterbricht der Seitenfehler-Handler den aktuellen Strom von Instruktionen im Übersetzer und führt die Steuerung zu einem vom Übersetzer vorgegebenen Prüfpunkt zurück.

Der Übersetzer liest Quelleninstruktionen in Einheiten, die Basisblöcke genannt werden. Wenn ein Seitenfehler eintritt, während er einen Basisblock liest, dann ignoriert der Übersetzer diesen Block, setzt jedoch die Übersetzung irgendwelcher anderen Blöcke fort. Nachdem alle Basisblöcke gelesen wurden, werden sie in einen Satz von Zielinstruktionen übersetzt. Das Verfahren des Ignorierens eines Basisblocks, der einen Seitenfehler verursacht, ist in Fig. 33 gezeigt. Bevor er einen Basisblock liest, legt der Übersetzer einen Prüfpunkt fest. Alle vor dem Prüfpunkt gelesenen Basisblöcke sind sicher und können von keinerlei Seitenfehlern beeinträchtigt werden, die nach dem Prüfpunkt geschehen. Dann versucht der Übersetzer, den nächsten Basisblock zu lesen. Wenn es einen Seitenfehler gibt, springt er sofort zum Prüfpunkt. Dadurch wird er veranlaßt, den Basisblock zu überspringen und zu versuchen, den nächsten zu lesen.

BESONDERE OBJEKTE DER SIEBENTEN AUSFÜHRUNGSFORM

Die Erfindung gemäß der siebenten Ausführungsform ist ein Weg, die Speicherzugriffskosten der dynamischen Übersetzung zu reduzieren, indem Seiten nicht in den physischen Speicher kopiert werden, während weiterhin gestattet wird, daß die Übersetzung fortgesetzt wird, wenn ein Speicherzugriff fehlschlägt.

ZUSAMMENFASSUNG DER SIEBENTEN AUSFÜHRUNGSFORM

Der beschriebene Mechanismus zum Beheben von Seitenfehlern ist ein Weg, die Kosten der dynamischen Übersetzung zu reduzieren, wenn auf einen nicht-physisch abgebildeten Speicher zugegriffen wird. Er ermöglicht, daß die dynamische Übersetzung fortgesetzt wird, auch wenn nicht alle der Quellenmaschineninstruktionen aufgrund von Seitenfehlern gelesen werden können. Die Erfindung ist ein Weg, die Speicherzugriffskosten der dynamischen Übersetzung zu reduzieren, indem Seiten nicht in den physischen Speicher kopiert werden, während weiterhin gestattet wird, daß die Übersetzung fortgesetzt wird, wenn ein Speicherzugriff fehlschlägt.

ACHTE AUSFÜHRUNGSFORM DER VORLIEGENDEN ERFINDUNG

AUFZEICHNUNG VON AUSGÄNGEN AUS DEM ÜBERSETZTEN CODE FÜR DIE DYNAMISCHE ÜBERSETZUNG

KURZFASSUNG DER ACHTEN AUSFÜHRUNGSFORM

Die dynamische Übersetzung ist ein Akt der Übersetzung eines Computerprogramms in einer Maschinensprache in eine andere Maschinensprache, während das Programm läuft. Der dynamische Übersetzer wählt die zu übersetzenden Instruktionen, indem er sie profiliert, während sie ausführen. Die häufig ausgeführten Instruktionen werden übersetzt, und die selten ausgeführten werden dies nicht. Die übersetzten Instruktionen können bewirken, daß der Profiler einige Instruktionen ausläßt, was dazu führen kann, daß häufig ausgeführte Instruktionen interpretiert werden. Durch die Aufzeichnung spezifischer Ausgänge aus dem übersetzten Code ist es möglich, alle der häufig ausgeführten Instruktionen zu profilieren und sicherzustellen, daß sie alle übersetzt werden.

BESCHREIBUNG VON FIGUREN DER ACHTEN AUSFÜHRUNGSFORM

Fig. 34 veranschaulicht ein Muster eines Steuerflusses in einem dynamischen Übersetzungssystem mit einem Ver-

zweigungsprofilierer gemäß einer achten Ausführungsform der vorliegenden Erfindung.

DETAILLIERTE BESCHREIBUNG DER ACHTEN AUSFÜHRUNGSFORM

Wie im Dokument 'Verzweigungsprotokollierer für die dynamische Übersetzung' beschrieben, profiliert das dynamische Übersetzungssystem die Verzweigungsinstruktionen des Originalprogramms, während sie interpretiert werden, um zu bestimmen, welche Instruktionen häufig ausgeführt werden und welche nicht. Der Verzweigungsprotokollierer profiliert nur Verzweigungsinstruktionen und verläßt sich auf die Annahme, daß alle häufig ausgeführten Instruktionen durch häufig ausgeführte Verzweigungen erreicht werden. In einigen Fällen macht der dynamische Übersetzer selbst diese Annahme un wahr, da die Steuerung von den übersetzten Instruktionen zurück zu den interpretierten Instruktionen fließt, ohne daß eine profilierte Verzweigung ausgeführt wird. Der Übersetzer kann diese Fälle identifizieren, und er schafft spezielle übersetzte Instruktionen, die diesen Steuerfluß profilieren als wäre er eine Verzweigung.

Fig. 34 veranschaulicht, wie die Steuerung von interpretierten Instruktionen zu übersetzten Instruktionen und zurück fließt. Wo immer die Steuerung einen Ausgang aus übersetzten Instruktionen vornimmt, stellt der Übersetzer sicher, daß der Ausgang profiliert wird als wäre er eine Verzweigungsinstruktion. Es gibt einige Fälle, in denen die Steuerung von übersetzten zu interpretierten Instruktionen fließt.

Erstens gibt es Verzweigungen zu nicht-festgelegten Zielen. Der Übersetzer weiß nicht, welche Instruktion nach der Verzweigung ausgeführt wird, daher kann er diese Instruktion nicht in dieselbe Übersetzungseinheit kombinieren wie die Verzweigung. Statt dessen schafft er einen Ausgang aus dem übersetzten Code zurück zum interpretierten Code.

Zweitens gibt es Instruktionen, die aufgrund von Seitenfehlern während der Übersetzung nicht gelesen werden können. Wie im Dokument 'Seitenfehlerbehebung für die dynamische Übersetzung' beschrieben, ignoriert der Übersetzer Blöcke von Instruktionen, die aufgrund eines Seitenfehlers nicht gelesen werden können. Daher muß das übersetzte Programm zu interpretierten Instruktionen zurückspringen, wenn es diese Blöcke erreicht.

Drittens werden einige Instruktionen selten ausgeführt, während die Übersetzung vorgenommen wird. Sie werden nicht übersetzt, da sie selten ausgeführt wurden, wie im Dokument 'Blockwahlschwelle für die dynamische Übersetzung' beschrieben. Sie können jedoch in der Zukunft häufiger ausgeführt werden, daher muß der Übersetzer Ausgänge zu diesen Instruktionen aufzeichnen. Dieses Merkmal ermöglicht es dem dynamischen Übersetzungssystem, sich an die Änderung von Ausführungsmustern anzupassen, welche die Verteilung häufig ausgeführter Instruktionen verändern.

Da die Ausgänge aus dem übersetzten Code aufgezeichnet werden, werden mehr Instruktionen übersetzt. Dies erhöht die Chance, daß eine übersetzte Version einer Instruktion existieren wird. Nachdem das dynamische Übersetzungssystem eine lange Zeit laufen gelassen wird, verursachen daher die meisten der Ausgänge aus einer übersetzten Einheit einen Sprung zu einer anderen übersetzten Einheit anstelle eines Sprungs zurück zum interpretierten Code. Dies hat einen direkten Vorteil durch die öftere Verwendung der schnelleren übersetzten Instruktionen und einen indirekten Vorteil durch die Ausführung der Verzweigungsprotokolliererinstruktionen nicht so oft.

BESONDERE OBJEKTE DER ACHTEN AUSFÜHRUNGSFORM

Die achte Ausführungsform der vorliegenden Erfindung ist auf ein Verfahren gerichtet, um sicherzustellen, daß häufig ausgeführte Instruktionen übersetzt werden, auch wenn sie nicht durch irgendwelche profilierte Verzweigungen erreicht werden, indem die möglichen Ausgänge aus übersetzten Instruktionseinheiten profiliert werden.

ZUSAMMENFASSUNG DER ACHTEN AUSFÜHRUNGSFORM

Ein dynamisches Übersetzungssystem muß alle häufig ausgeführten Instruktionen lokalisieren und übersetzen, was durch Profilierverzweigungsinstruktionen erzielt werden kann. Die Übersetzung von Instruktionen wird jedoch Pfade zu Instruktionen schaffen, die keine profilierten Verzweigungen enthalten. Daher wird die Profilierung erweitert, um die Ausgänge aus übersetzten Instruktionen zu enthalten. Die Erfindung ist ein Verfahren, um sicherzustellen, daß häufig ausgeführte Instruktionen übersetzt werden, auch wenn sie nicht durch irgendwelche profilierte Verzweigungen erreicht werden, indem die möglichen Ausgänge aus übersetzten Instruktionseinheiten profiliert werden.

NEUNTE AUSFÜHRUNGSFORM DER VORLIEGENDEN ERFINDUNG

BLOCKWAHLSCHWELLE FÜR DIE DYNAMISCHE ÜBERSETZUNG

KURZFASSUNG DER NEUNTEN AUSFÜHRUNGSFORM

Die dynamische Übersetzung ist ein Akt der Übersetzung eines Computerprogramms in einer Maschinensprache in eine andere Maschinensprache, während das Programm läuft. Der dynamische Übersetzer sollte alle der häufig ausgeführten Teile des Quellenprogramms übersetzen und alle der selten ausgeführten Teile ignorieren. Um dies zu erzielen, profiliert das Übersetzungssystem Verzweigungsinstruktionen und übersetzt jene Instruktionen nicht, deren Ausführungswahrscheinlichkeit unter einer spezifizierten Schwelle liegt.

BESCHREIBUNG VON FIGUREN DER NEUNTEN AUSFÜHRUNGSFORM

Fig. 35 veranschaulicht, wie der dynamische Übersetzer Verzweigungsprofilinformationen verwendet, um die Ausführungswahrscheinlichkeit eines Basisblocks zu berechnen, gemäß einer neunten Ausführungsform der vorliegenden Erfindung.

Der Zweck eines dynamischen Übersetzters ist, die allgemeine Ausführungsgeschwindigkeit eines Computerprogramms zu verbessern, indem er es aus seinen ursprünglichen Quellen-Sprachinstruktionen in effizientere Ziel-Sprachinstruktionen übersetzt. Der Vorteil der dynamischen Übersetzung wird durch den Vergleich der Gesamtzeit für die Ausführung des Originalprogramms mit der für die Übersetzung des Programms erforderlichen Zeit plus der Zeit für die Ausführung des übersetzten Programms gemessen. Die zur Übersetzung irgendeines Teils des Programms erforderliche Zeit ist ungefähr konstant, daher wird der Vorteil der Übersetzung eines Teils hauptsächlich durch die Anzahl von Malen, die dieser Teil verwendet wird, bestimmt. Häufig ausgeführte Instruktionen sind es Wert, übersetzt zu werden, selten ausgeführte Instruktionen sind es jedoch nicht Wert, übersetzt zu werden.

Zur Messung der Frequenz verschiedener Instruktionen kann ein dynamisches Übersetzungssystem Verzweigungsstrukturen profilieren. Unter Verwendung dieser Profilinformatoren kann es eine häufig ausgeführte Instruktion wählen und an diesem Punkt die Übersetzung beginnen. Nach der anfänglichen Instruktion versucht der Übersetzer so viele häufig ausgeführte Nachfolgerinstruktionen zu lesen wie möglich, ohne die seltenen Nachfolger zu lesen. Die Blockwahrschwelle wird verwendet, um zu bestimmen, ob ein Nachfolger häufig oder selten ausgeführt wird.

Der dynamische Übersetzer liest Instruktionen in Einheiten, die Basisblöcke genannt werden. In einem Basisblock werden alle der Instruktionen die gleiche Anzahl von Malen ausgeführt, daher werden entweder alle häufig ausgeführt oder alle selten ausgeführt.

Der dynamische Übersetzer verwendet Profilinformatoren aus Verzweigungsstrukturen, um zu bestimmen, ob ein Basisblock häufig oder selten ausgeführt wird. Dieser Prozeß ist in Fig. 35 gezeigt. Der Übersetzer berechnet die Wahrscheinlichkeit, daß ein Ausführungspfad von der ersten übersetzten Instruktion zu einem gegebenen Basisblock eingeschlagen wird. Der erste Basisblock erhält eine Ausführungswahrscheinlichkeit von 100%, da er die erste Instruktion enthält. Wenn der aktuelle Block nur einen Nachfolger hat, dann hat der Nachfolger dieselbe Ausführungswahrscheinlichkeit wie der aktuelle Block. Wenn der aktuelle Block in einer bedingten Verzweigung endet, dann wird die Wahrscheinlichkeit des aktuellen Blocks zwischen den beiden Nachfolgern gemäß den Verzweigungsprofilinformationen geteilt. Wenn die Ausführungswahrscheinlichkeit des aktuellen Blocks beispielsweise 50% war, und er in einer Verzweigungsstrukturen endet, die 40 Mal ausgeführt und 10 Mal eingeschlagen wurde, dann wäre die Wahrscheinlichkeit des Verzweigung-ingeschlagen-Nachfolgers ($50\% \cdot 25\% = 12,5\%$), und die Wahrscheinlichkeit des Fehlschlag-Nachfolgers wäre ($50\% \cdot 75\% = 37,5\%$).

Eine variable Schwelle, die Blockwahrschwelle genannt wird, wird zum Auswählen häufig ausgeführter Blöcke verwendet. Wenn die Ausführungswahrscheinlichkeit eines Blocks größer als die oder gleich der Schwelle ist, dann wird dieser Block als häufig ausgeführt angesehen, und er wird übersetzt. Wenn die Ausführungswahrscheinlichkeit unter der Schwelle liegt, dann wird der Block als selten ausgeführt angesehen, und er wird nicht übersetzt.

Eine wichtige Eigenschaft dieses Blockwahlverfahrens ist, daß der Satz gewählter Blöcke verbunden ist. Es gibt kompliziertere Wege der Berechnung der Ausführungswahrscheinlichkeit, wie das Addieren der Wahrscheinlichkeiten aus allen Vorläufern. Dies kann jedoch zu nicht-zusammenhängenden Sätzen von Blöcken führen. Es ist möglich, nicht-zusammenhängende Sätze von Blöcken zu übersetzen, aber es gibt mehr Möglichkeiten zur Optimierung des übersetzten Codes, wenn er völlig zusammenhängend ist.

BESONDERE OBJEKTE DER NEUNTEN AUSFÜHRUNGSFORM

Die neunte Ausführungsform der vorliegenden Erfindung ist auf ein Verfahren zur Verbesserung der Effizienz der dynamischen Übersetzung gerichtet, indem Blöcke häufig ausgeführter Instruktionen zur Übersetzung gewählt und Blöcke selten ausgeführter Instruktionen ignoriert werden, wobei eine Schwellenausführungswahrscheinlichkeit zur Trennung der häufig ausgeführten Blöcke von selten ausgeführten verwendet wird.

ZUSAMMENFASSUNG DER NEUNTEN AUSFÜHRUNGSFORM

Ein dynamisches Übersetzungssystem muß kostenproportional zur Anzahl übersetzter Instruktionen und nutzenproportional zur Anzahl von Malen sein, die eine übersetzte Instruktion ausgeführt wird. Daher ist es am effizientesten, nur häufig ausgeführte Instruktionen zu übersetzen, und die selten ausgeführten zu ignorieren. Die Erfindung ist ein Verfahren zur Verbesserung der Effizienz der dynamischen Übersetzung, indem Blöcke häufig ausgeführter Instruktionen zur Übersetzung gewählt und Blöcke selten ausgeführter Instruktionen ignoriert werden, wobei eine Schwellenausführungswahrscheinlichkeit zur Trennung der häufig ausgeführten Blöcke von selten ausgeführten verwendet wird.

Obwohl einige bevorzugte Ausführungsformen der vorliegenden Erfindung erläutert und beschrieben wurden, ist es für Fachleute klar, daß Änderungen in diesen Ausführungsformen vorgenommen werden können, ohne von den Prinzipien und dem Grundgedanken der Erfindung abzuweichen, deren Umfang in den Ansprüchen und ihren Äquivalenten definiert ist.

Patentansprüche

1. Computerarchitektur-Emulationssystem, welches eine Quellen-Computerarchitektur auf einer Ziel-Computerarchitektur emuliert, mit:
einer Interpretierereinrichtung zum individuellen Übersetzen eines Quellen-Objektcodes in einen entsprechenden übersetzten Objektcode und zum Bestimmen einer Anzahl von Ausführungen von Verzweigungsstrukturen im Quellen-Objektcode; und
einer Kompilierereinrichtung zum Gruppieren von Instruktionen des Quellen-Objektcodes in ein Segment, wenn eine Anzahl von Ausführungen einer entsprechenden Verzweigungsstrukturen eine Schwellenanzahl überschreitet,

und zum dynamischen Kompilieren des Segments.

2. Computerarchitektur-Emulationssystem nach Anspruch 1, bei welchem Verzweigungs-Objektcode-Instruktionen, die Segmenten entsprechen, welche nicht kompiliert sind, im Speicher gespeichert werden.

3. Computerarchitektur-Emulationssystem nach Anspruch 2, bei welchem Segmente, die Verzweigungs-Objektcode-Instruktionen entsprechen, welche die Schwellenanzahl nicht überschritten haben, nicht kompiliert werden.

4. Computerarchitektur-Emulationssystem nach Anspruch 1, bei welchem Segmente, die Verzweigungs-Objektcode-Instruktionen entsprechen, welche Segmenten entsprechen, die nicht kompiliert sind, im Speicher gespeichert werden, während die Interpretiereinrichtung die übersetzten Objektcode-Instruktionen ausführt.

5. Computerarchitektur-Emulationssystem nach Anspruch 1, bei welchem die Interpretiereinrichtung und die Kompilierereinrichtung Aufgaben sind, die gleichzeitig in einem Mehraufgaben-Betriebssystem in Echtzeit operieren.

6. Computerarchitektur-Emulationssystem nach Anspruch 1, ferner mit:
einer Verzweigungsprotokolliereinrichtung zum Speichern von Verzweigungsprofilinformationen der Verzweigungsinstruktionen, die von der Interpretiereinrichtung bestimmt werden.

7. Computerarchitektur-Emulationssystem nach Anspruch 6, bei welchem
die Verzweigungsprofilinformationen eine Verzweigungsadresse, einen Verzweigungsnachfolger, einen Nicht-Verzweigungsnachfolger, eine Verzweigungsausführungszählung und eine Verzweigung-eingeschlagen-Zählung enthalten, und

die Verzweigungsprofilinformationen von der Interpretiereinrichtung während der Verzweigungsinstruktions-emulation protokolliert werden.

8. Computerarchitektur-Emulationssystem nach Anspruch 1, ferner mit:
einer Einrichtung zum Plazieren einer Codeflagge nach Verzweigungsinstruktionen, die einen Sprung in übersetzbare Instruktionen oder aus diesen ausführen; und

einer Einrichtung zum Prüfen, ob Nachfolgerinstruktionen der entsprechenden Verzweigungsinstruktionen übersetzbar sind oder nicht, indem auf die entsprechende Codeflagge bezuggenommen wird.

9. Computerarchitektur-Emulationssystem nach Anspruch 1, ferner mit:
einer Einrichtung zum Initiieren einer Verzweigungsinstruktion, wenn eine Anzahl von Ausführungen einer Nachfolgerinstruktion der Verzweigungsinstruktion einen Schwellenwert übersteigt.

10. Computerarchitektur-Emulationssystem nach Anspruch 1, ferner mit:
einer Einrichtung zum Kommunizieren zwischen der Interpretiereinrichtung und der Kompilierereinrichtung, während die Interpretiereinrichtung die Emulation des Quellencodes fortsetzt, um die Übersetzung von Segmenten, die häufig verzweigten Instruktionen entsprechen, zu initiieren.

11. Computerarchitektur-Emulationssystem nach Anspruch 1, ferner mit:
einer Einrichtung zum Steuern einer Kompilierungsrate von zu kompilierenden Segmenten durch die Erhöhung der Schwellenanzahl, wenn eine Warteschlange zum Speichern der zu übersetzenden Segmente eine vorherbestimmte Kapazität erreicht.

12. Computerarchitektur-Emulationssystem nach Anspruch 1, bei welchem die Kompilierereinrichtung ein optimiertes Objekt erzeugt, während jede Instruktion, die sich im Speicher befindet, der Reihe nach verfolgt wird, indem ein Profil verwendet wird, das der Adresse entspricht, an der die Kompilierung gestartet wurde.

13. Computerarchitektur-Emulationssystem nach Anspruch 12, bei welchem die Kompilierereinrichtung einen Block bei der Detektion eines Seitenfehlers nicht kompiliert, so daß, wenn ein Block einen Seitenfehler verursacht, die Kompilierereinrichtung ein Objekt produziert, um Verzweigungsinformationen in der Verzweigungsprotokolliereinrichtung zu protokollieren.

14. Computerarchitektur-Emulationssystem nach Anspruch 13, bei welchem, wenn ein Instruktionsausführungsprozeß in bezug auf eine vorherbestimmte Rate nicht rechtzeitig ausführt, die Kompilierereinrichtung die Ausführung unter Verwendung eines Profils verfolgt, prüft, ob eine Verzweigungszählung unter einer vorherbestimmten Anzahl liegt, und ein Objekt produziert, um Verzweigungsinformationen zu protokollieren.

15. Computerarchitektur-Emulationssystem nach Anspruch 1, ferner mit:
einer Verzweigungsprotokolliereinrichtung zum Speichern von Profilinformationen der Verzweigungsinstruktionen im Quellen-Objektcode, der die Anzahl von Ausführungen enthält, wobei die Verzweigungsprotokolliereinrichtung einen Cache zum Speichern von Profilinformationen häufig ausgeführter Verzweigungsinstruktionen und ein Verzweigungsprotokoll zum Speichern von Profilinformationen weniger häufig ausgeführter Verzweigungsinstruktionen enthält.

16. Computerarchitektur-Emulationssystem nach Anspruch 15, bei welchem die Profilinformationen im Cache durch das Kombinieren von Verzweigungsadresseninformationen und Verzweigungszielinformationen organisiert werden.

17. Computerarchitektur-Emulationssystem nach Anspruch 16, bei welchem die im Cache organisierten Profilinformationen in einer Vielzahl von Gruppen in absteigender Reihenfolge des Eintrags in die Gruppe gespeichert werden.

18. Computerarchitektur-Emulationssystem nach Anspruch 1, bei welchem jede Verzweigungsinstruktion ein Startparameter ist, wobei die Kompilierereinrichtung ferner enthält:
einen Blockwähler, der ein Segment des zu kompilierenden Quellen-Objektcodes auf der Basis des Startparameters und der Profilinformationen der Verzweigung auswählt,

eine Blockaufbaueinheit, die das Segment in eine lineare Liste von Instruktionen abflacht, und

eine optimierende Codegenerierungseinheit, welche die tatsächliche Kompilierung von Originalinstruktionen in übersetzte Codesegmentinstruktionen durchführt.

19. Computerarchitektur-Emulationssystem nach Anspruch 18, bei welchem der Blockwähler einen Steuerflußgraphen schafft, der die zu kompilierenden Originalinstruktionen beschreibt, und den Steuerflußgraphen zur Blockauf-

baueinheit weiterreicht.

20. Computerarchitektur-Emulationssystem, welches eine Quellen-Computerarchitektur auf einem Ziel-Computerarchitektursystem emuliert, mit:

einer Vielzahl von Interpretierereinrichtungen zum individuellen Übersetzen eines Quellen-Objektcodes in einen entsprechenden Übersetzten Objektcode, wobei jede der Vielzahl von Interpretierereinrichtungen Quellen-Objektcode-Verzweigungsinformationen in Echtzeit profiliert, während sie übersetzte Objektcode-Instruktionen ausführt; und

einer Kompilierereinrichtung zum Gruppieren von Quellen-Objektcode-Instruktionen von irgendeiner der Vielzahl von Interpretierereinrichtungen in Segmente auf der Basis entsprechender Verzweigungsinstruktionen im Quellen-Objektcode und zum dynamischen Kompilieren der Segmente des Quellen-Objektcodes, wenn die entsprechende Verzweigungsinstruktion größer ist als eine Schwellenanzahl.

21. Computerarchitektur-Emulationssystem nach Anspruch 20, bei welchem jede der Vielzahl von Interpretierereinrichtungen die Verzweigungs-Objektcode-Instruktionen profiliert und die Verzweigungs-Objektcode-Instruktionen speichert, welche die Schwellenanzahl nicht überschritten haben, indem ein Verzweigungsprotokollierer aufgerufen wird.

22. Computerarchitektur-Emulationssystem, welches eine Quellen-Computerarchitektur auf einem Ziel-Computerarchitektursystem emuliert, mit:

einer Interpretierereinrichtung zum individuellen Übersetzen eines Quellen-Objektcodes in einen entsprechenden übersetzten Objektcode, wobei die Interpretierereinrichtung Verzweigungsinstruktionen des Quellen-Objektcodes durch das Speichern einer Anzahl von Ausführungen für jede Verzweigungsinstruktion und Vergleichen der Anzahl von Ausführungen mit einer Schwellenanzahl profiliert, so daß Verzweigungsinstruktionen, welche die Schwellenanzahl überschreiten, Startparameter sind; und

einer Kompilierereinrichtung zum Gruppieren der Quellen-Objektcode-Instruktionen in Segmente auf der Basis der Startparameter und dynamischen Kompilieren der Segmente des Quellen-Objektcodes während der Übersetzung und Profilierung durch die Interpretierereinrichtung.

23. Computerarchitektur-Emulationssystem nach Anspruch 22, bei welchem jedes Segment Instruktionen enthält, die aus der Optimierung des Quellen-Objektcodes auf der Basis eines entsprechenden Startparameters resultieren, und jedes Segment als Einheit installiert und deinstalliert wird.

24. Computerarchitektur-Emulationssystem nach Anspruch 23, bei welchem Verzweigungs-Objektcode-Instruktionen, die Segmenten entsprechen, welche nicht kompiliert sind, im Speicher gespeichert werden, während Segmente, die Verzweigungs-Objektcode-Instruktionen entsprechen, welche die Schwellenanzahl nicht überschritten haben, nicht kompiliert werden.

25. Computerarchitektur-Emulationssystem nach Anspruch 23, ferner mit:

einer Verzweigungsprotokollierereinrichtung zum Speichern von Verzweigungsprofilinformationen der Verzweigungsinstruktionen, die von der Interpretierereinrichtung bestimmt werden, wobei die Verzweigungsprofilinformationen eine Verzweigungsadresse, einen Verzweigungsnachfolger, einen Nicht-Verzweigungsnachfolger, eine Verzweigungsausführungszählung und eine Verzweigung-eingeschlagen-Zählung enthalten, und die Verzweigungsprofilinformationen von der Interpretierereinrichtung während der Verzweigungsinstruktionsemulation protokolliert werden.

26. Computerarchitektur-Emulationssystem nach Anspruch 23, ferner mit:

einer Einrichtung zum Plazieren einer Codeflagge nach Verzweigungsinstruktionen, die einen Sprung in übersetzbare Instruktionen oder aus diesen ausführen; und

einer Einrichtung zum Prüfen, ob Nachfolgerinstruktionen der entsprechenden Verzweigungsinstruktionen übersetzbar sind oder nicht, indem auf die entsprechende Codeflagge bezuggenommen wird.

27. Computerarchitektur-Emulationssystem nach Anspruch 23, ferner mit:

einer Einrichtung zum Initiieren einer Verzweigungsinstruktion, wenn eine Anzahl von Ausführungen einer Nachfolgerinstruktion der Verzweigungsinstruktion einen Schwellenwert übersteigt.

28. Computerarchitektur-Emulationssystem nach Anspruch 23, ferner mit:

einer Einrichtung zum Steuern einer Kompilierungsrate von zu kompilierenden Segmenten durch die Erhöhung der Schwellenanzahl, wenn eine Warteschlange zum Speichern der zu übersetzenden Segmente eine vorherbestimmte Kapazität erreicht.

29. Computerarchitektur-Emulationssystem nach Anspruch 23, bei welchem, wenn ein Instruktionsausführungsprozeß in bezug auf eine vorherbestimmte Rate nicht rechtzeitig ausführt, die Kompilierereinrichtung die Ausführung unter Verwendung eines Profils verfolgt, prüft, ob eine Verzweigungszählung unter einer vorherbestimmten Anzahl liegt, und ein Objekt produziert, um Verzweigungsinformationen wie den Seitenfehler zu protokollieren.

30. Computerarchitektur-Emulationssystem nach Anspruch 23, ferner mit:

einer Verzweigungsprotokollierereinrichtung zum Speichern von Profilinformationen der Verzweigungsinstruktionen im Quellen-Objektcode, der die Anzahl von Ausführungen enthält,

wobei die Verzweigungsprotokollierereinrichtung einen Cache zum Speichern von Profilinformationen häufig ausgeführter Verzweigungsinstruktionen und ein Verzweigungsprotokoll zum Speichern von Profilinformationen weniger häufig ausgeführter Verzweigungsinstruktionen enthält,

wobei die Profilinformationen im Cache durch das Kombinieren von Verzweigungsadresseninformationen und Verzweigungszielinformationen organisiert werden, und die im Cache organisierten Profilinformationen in einer Vielzahl von Gruppen in absteigender Reihenfolge des Eintrags in die Gruppe gespeichert werden.

31. Computerarchitektur-Emulationssystem nach Anspruch 23, bei welchem die Kompilierereinrichtung ferner enthält:

einen Blockwähler, der ein Segment des zu kompilierenden Quellen-Objektcodes auf der Basis des Startparameters

und der Profilinformatoren der Verzweigung auswählt, wobei der Blockwähler einen Steuerflußgraphen schafft, der die zu kompilierenden Originalinstruktionen beschreibt;
eine Blockaufbaueinheit, die den Steuerflußgraphen in eine lineare Liste von Instruktionen abflacht, und
eine optimierende Codegenerierungseinheit, welche die tatsächliche Kompilierung von Originalinstruktionen in übersetzte Codesegmentinstruktionen durchführt.

32. Mehraufgaben-Computerarchitektur-Emulationssystem, welches eine Quellen-Computerarchitektur auf einer Mehraufgaben-Ziel-Computerarchitektur emuliert, mit:

einer Interpretiereraufgabe zum individuellen Übersetzen eines Quellen-Objektcodes in einen entsprechenden übersetzten Objektcode und zum Bestimmen einer Anzahl von Ausführungen von Verzweigungsinstruktionen im Quellen-Objektcode; und

einer Kompiliereraufgabe, die mit dem Interpretierer auf der Mehraufgaben-Ziel-Computerarchitektur operiert, zum Gruppieren von Instruktionen des Quellen-Objektcodes in ein Segment, wenn eine Anzahl von Ausführungen einer entsprechenden Verzweigungsinstruktion eine Schwellenanzahl überschreitet, und zum dynamischen Kompilieren des Segments.

33. Mehraufgaben-Computerarchitektur-Emulationssystem nach Anspruch 32, welches Mehraufgaben-Computerarchitektur-Emulationssystem ein dynamisches Übersetzungssystem ist, wobei das Mehraufgaben-Computerarchitektur-Emulationssystem ferner umfaßt:

eine Software-Rückkopplungseinrichtung zum Ausgleichen einer Rate von Kompilierungsanforderungen, die von der Interpretiereraufgabe gesendet werden, und der Rate von Kompilierungen, die von der Kompiliereraufgabe vollendet werden, ohne daß es der Kompiliereraufgabe gestattet wird, untätig zu werden, indem die Schwellenanzahl variiert wird.

34. Mehraufgaben-Computerarchitektur-Emulationssystem nach Anspruch 33, ferner mit:

einer Warteschlange zum Speichern von Segmenten, die von der Kompiliereraufgabe zu kompilieren sind, wobei die Schwellenanzahl mit einer Mindestschwellenanzahl verglichen wird, um die Kompiliereraufgabe ein- oder aus zu schalten.

Hierzu 19 Seite(n) Zeichnungen

- Leerseite -

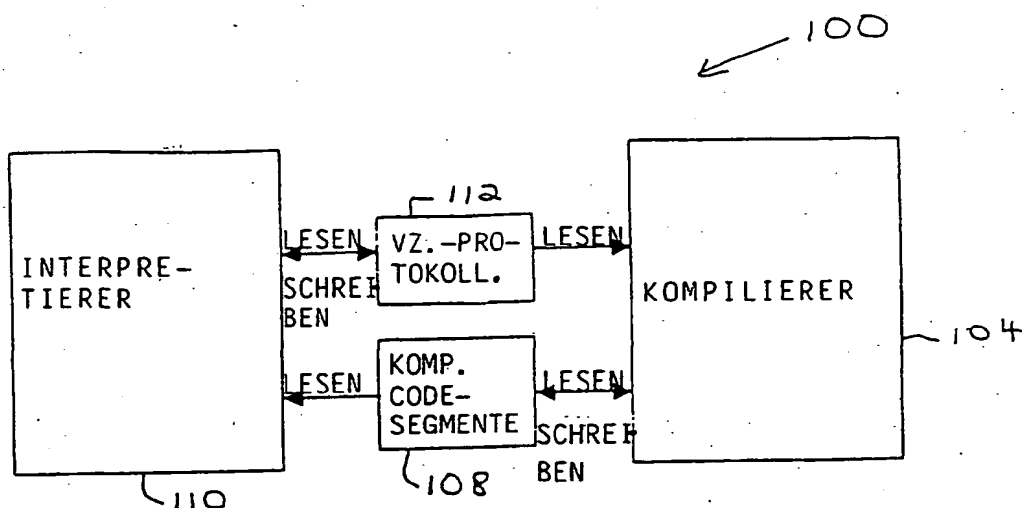


FIG. 1

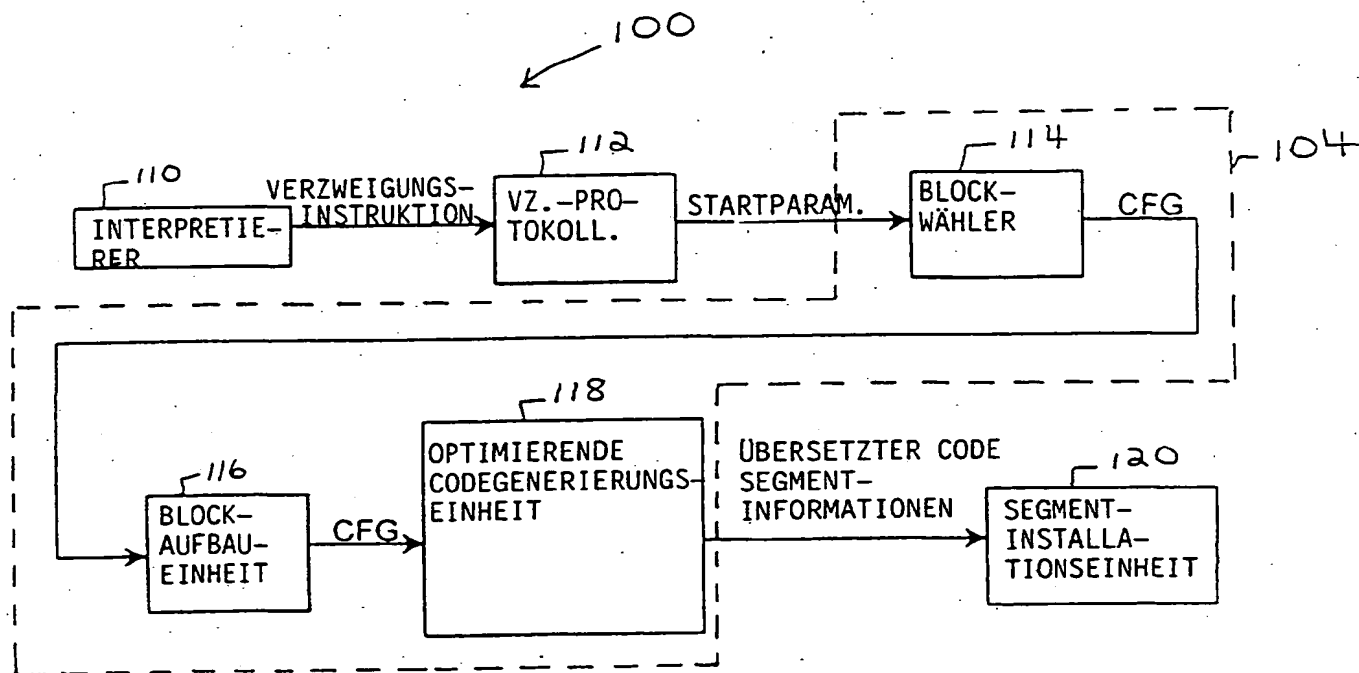


FIG. 2

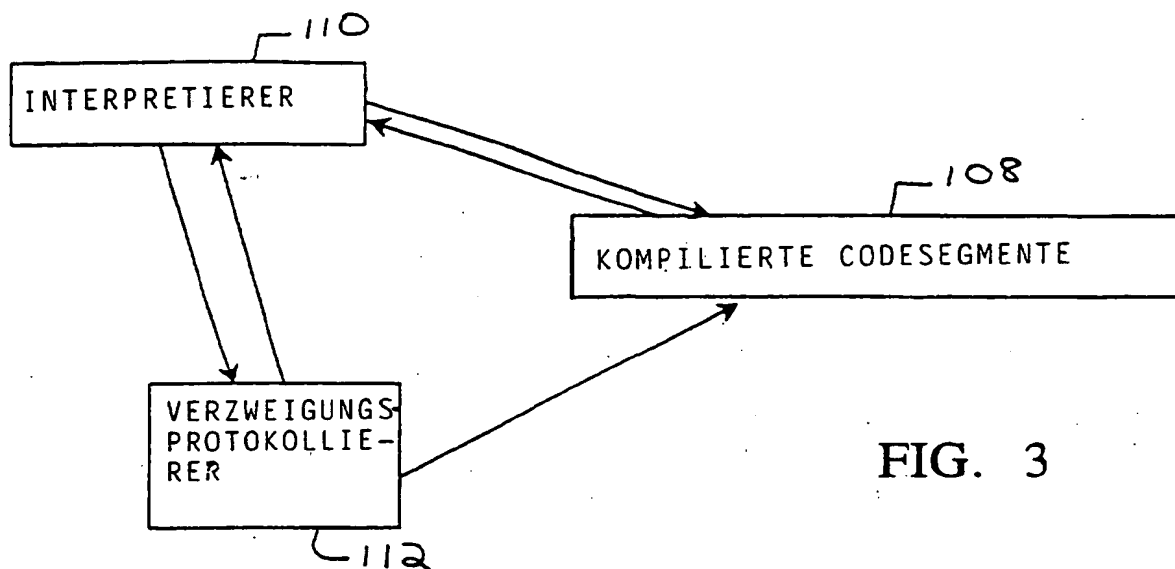


FIG. 3

FIG. 4

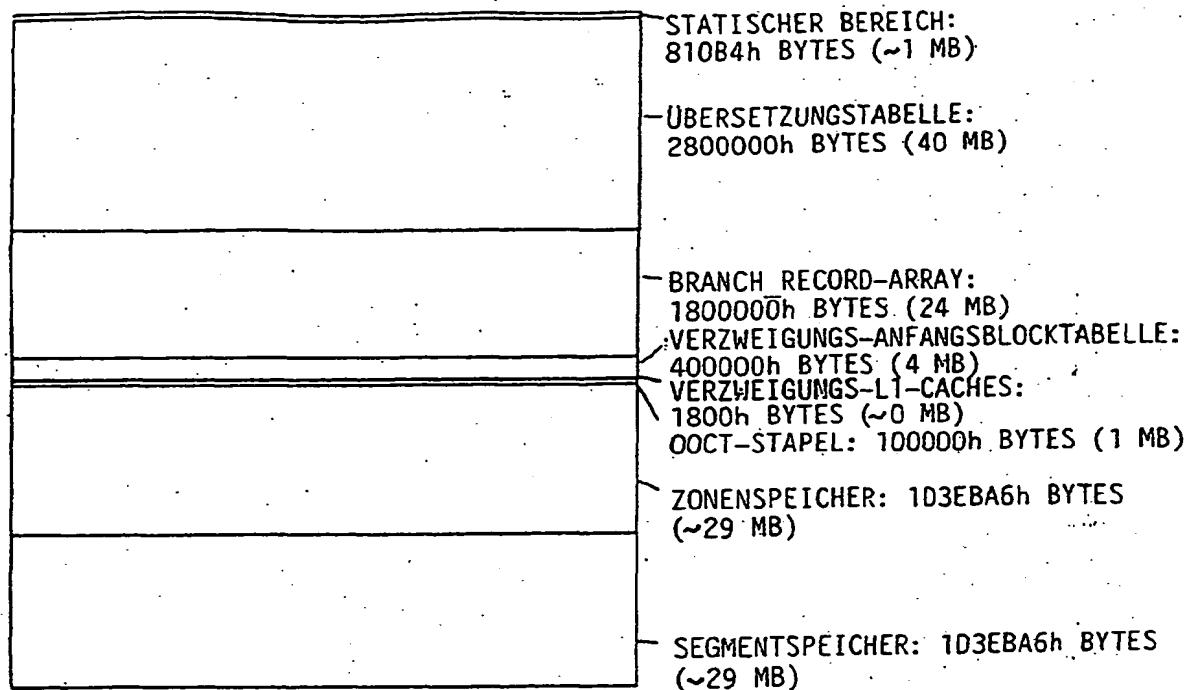


FIG. 5a

ASP-ADRESSE

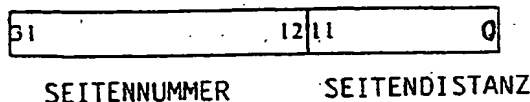


FIG. 5b

INDEXIERT
DURCH
SEITENNUMMER

ÜBERSETZUNGSTABELLE
ERSTER ORDNUNG

VOM ASP V. SEITE
NICHT VERWENDET
NICHT VERWENDET
VOM ASP V. SEITE
NICHT VERWENDET

ÜBERSETZUNGSTABELLE
ZWEITER ORDNUNG

EINTR.F.D. 0
EINTR.F.D. 1
...
ETR.F.D. Offfffh

INDEXIERT
DURCH
SEITENDISTANZ

ÜBERSETZUNG
ZWEITER
ORDNUNG
TABELLENEINTRAG

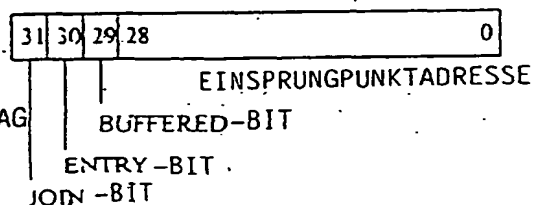
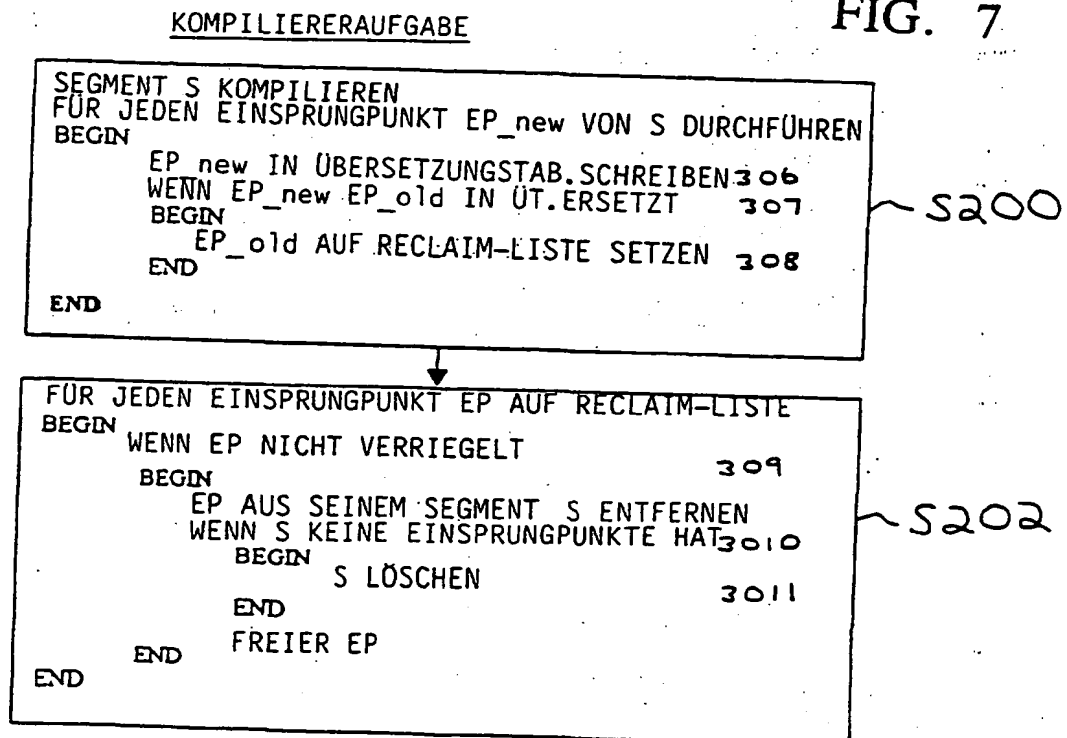
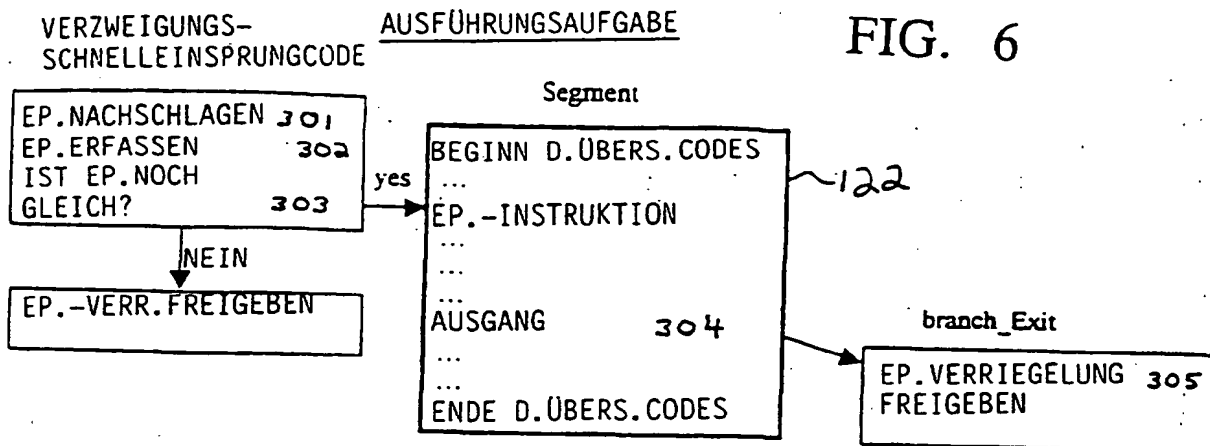


FIG. 5c



branch_address
branch_destination
branch_fall_through
encountered_count
taken_count
next

FIG. 8

local_branch_header_table

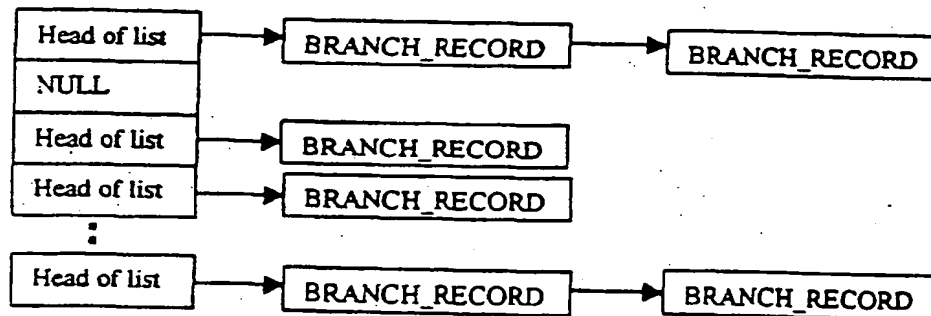


FIG. 9

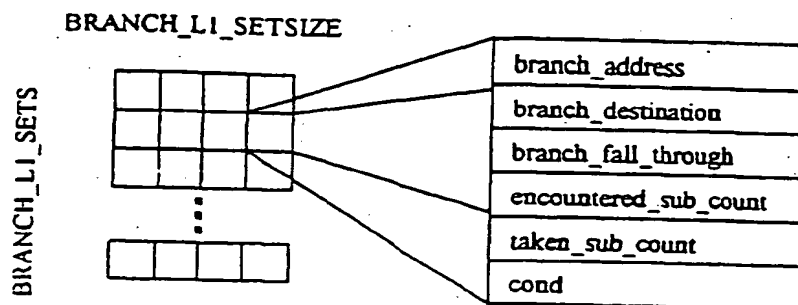


FIG. 10

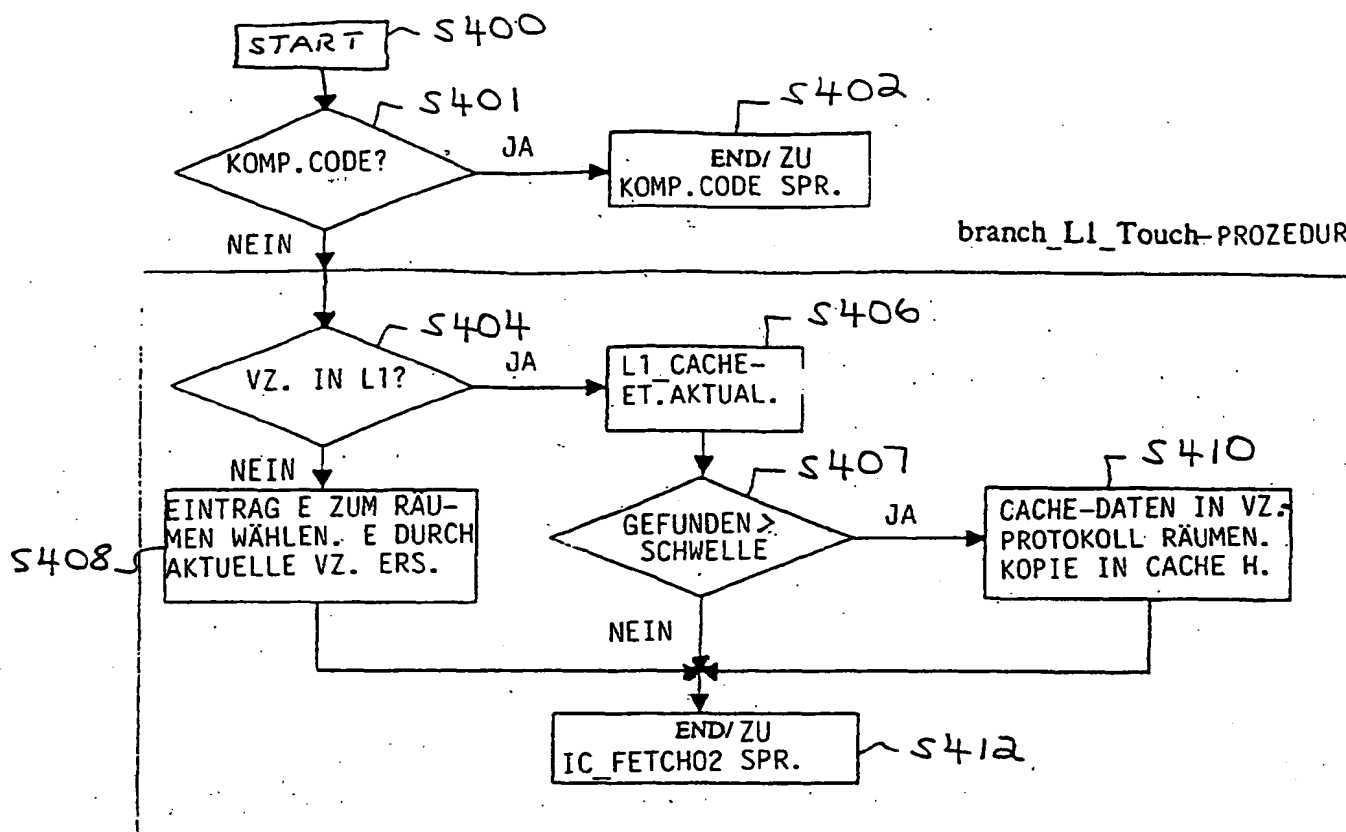


FIG. 11

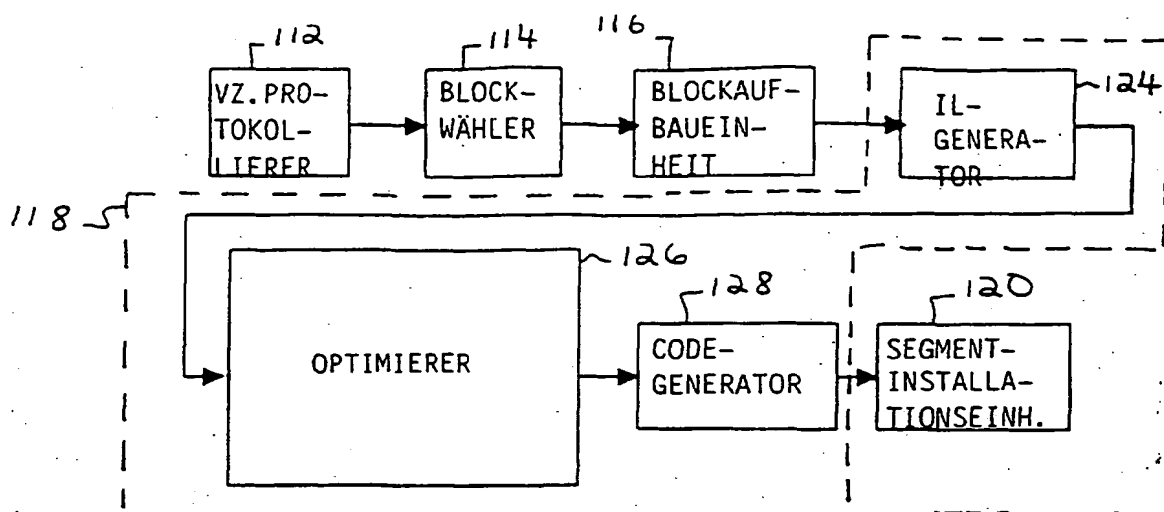


FIG. 12

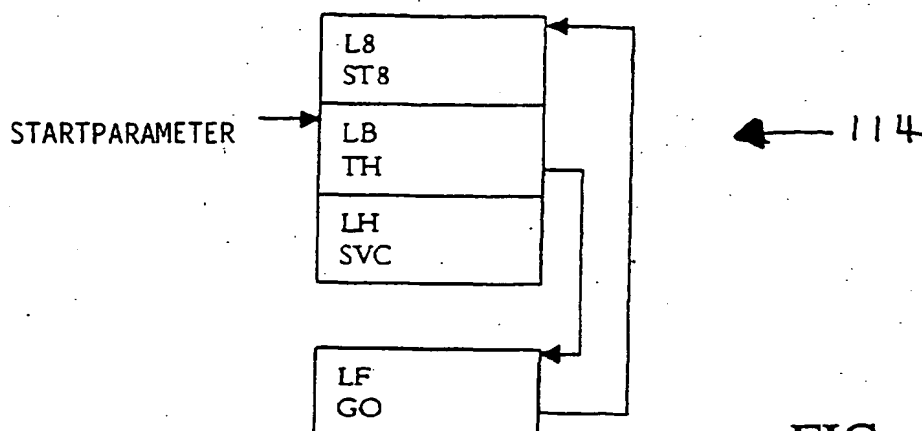


FIG. 13

```

LABEL L1
...
LABEL L2
...
IGOTO
ENTRY 1
  ASSIGN r1 r35
  ASSIGN r2 r36
  GOTO L1
ENTRY 2
  ASSIGN r1 r35
  GOTO L2
  
```

FIG. 14

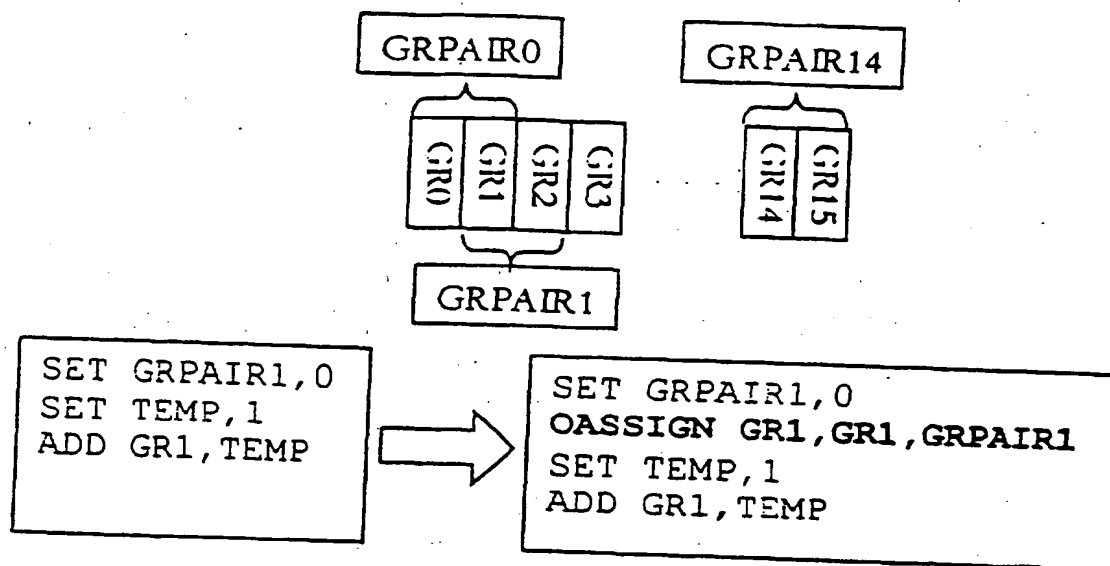


FIG. 15

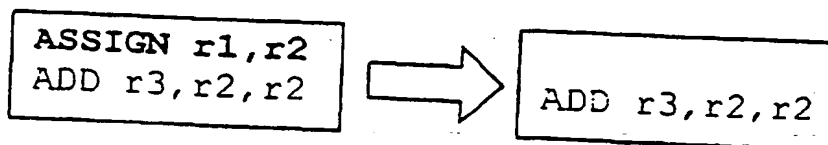


FIG. 16

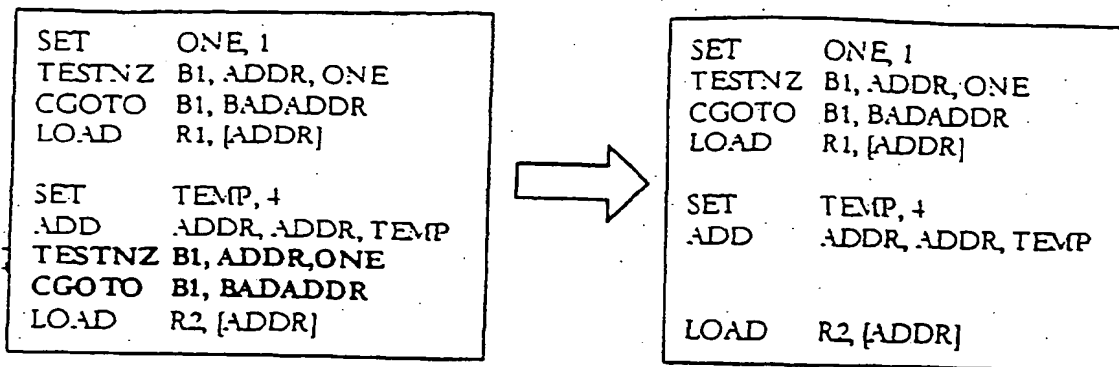


FIG. 17

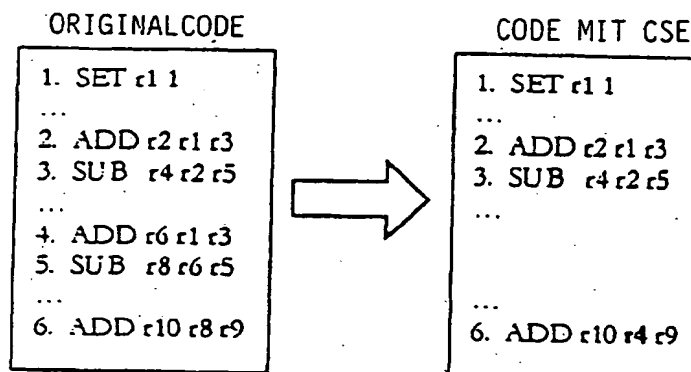


FIG. 18

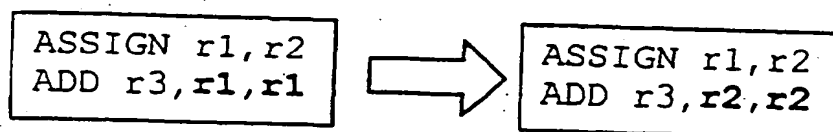


FIG. 19

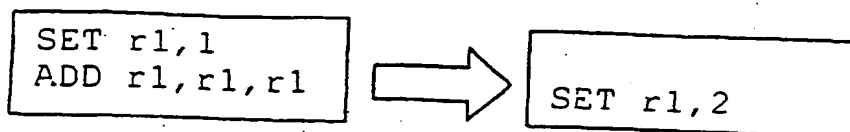


FIG. 20

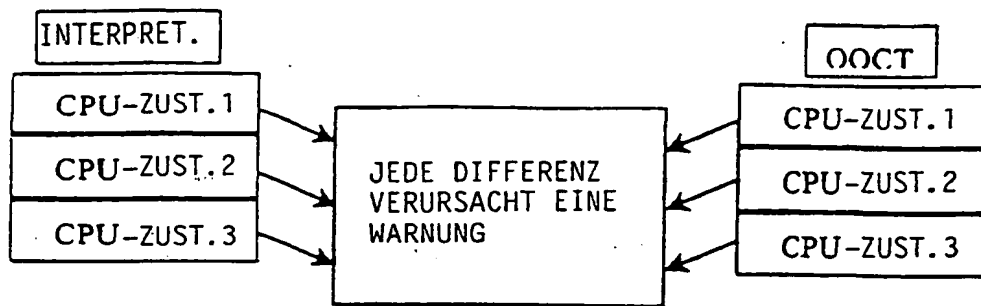


FIG. 21

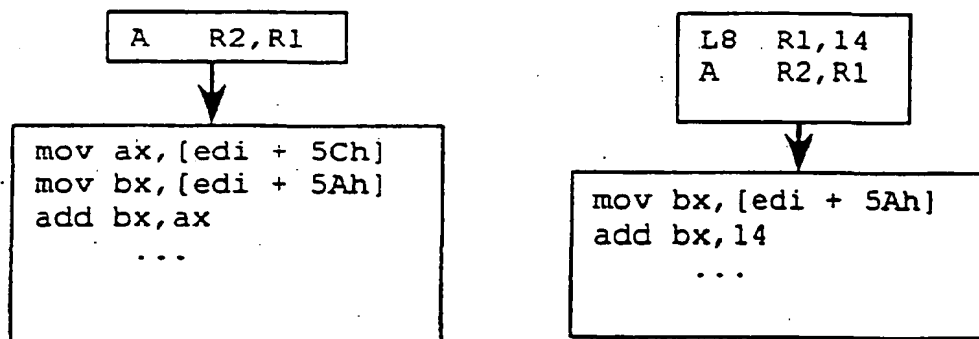


FIG. 22

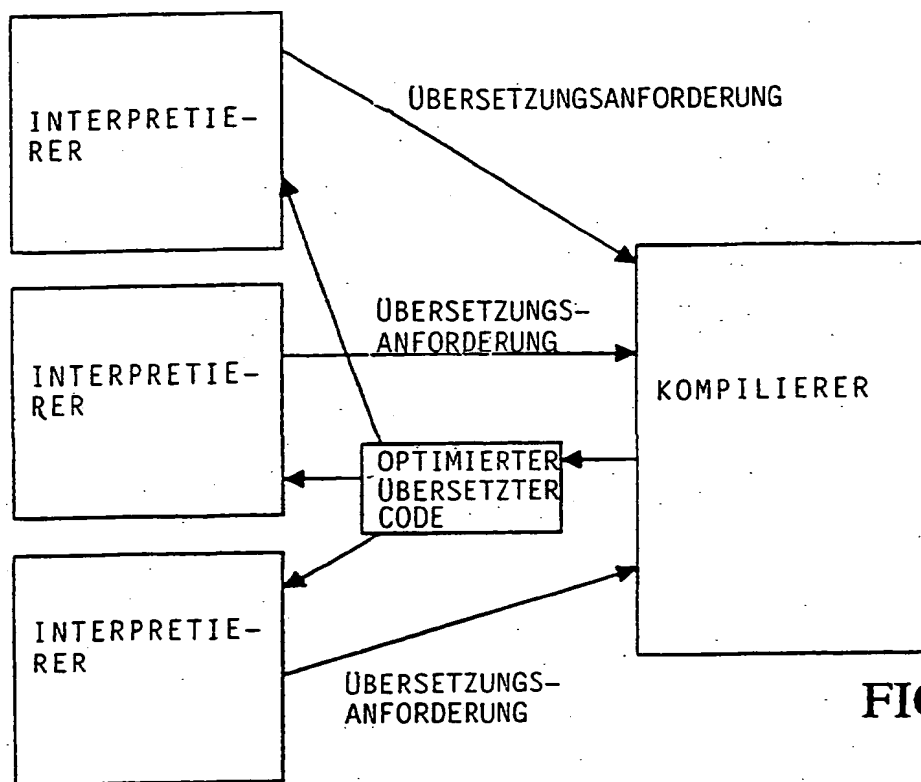


FIG. 23

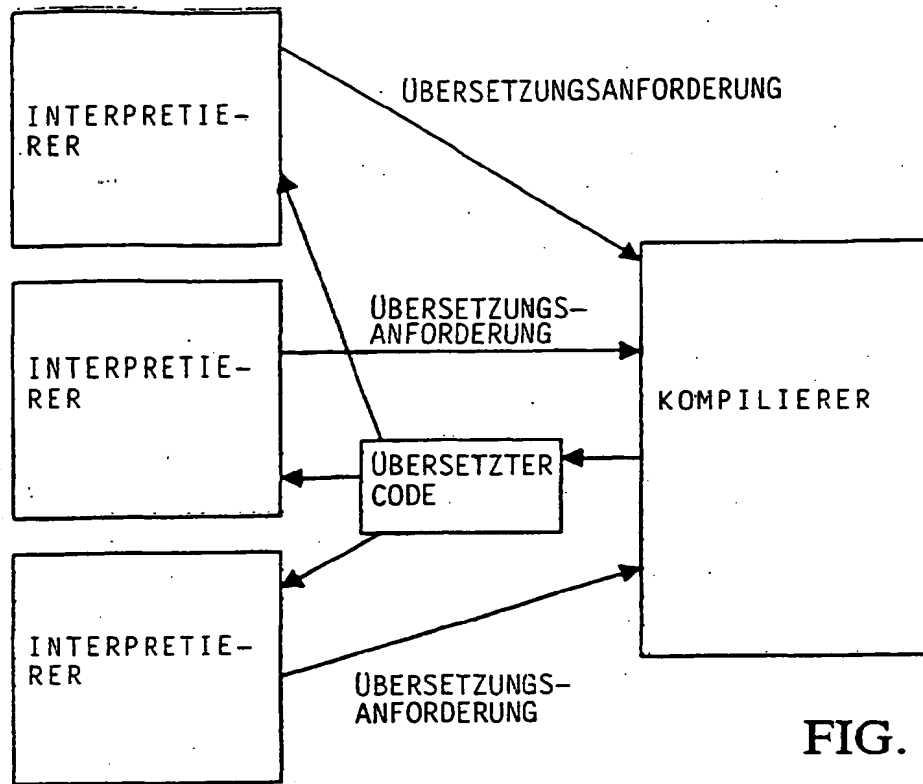


FIG. 24

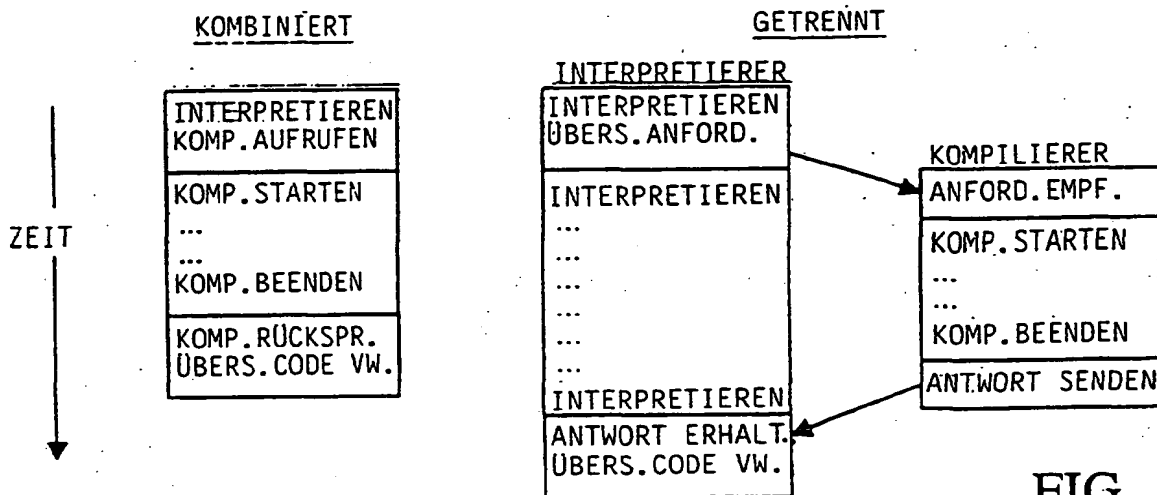


FIG. 25

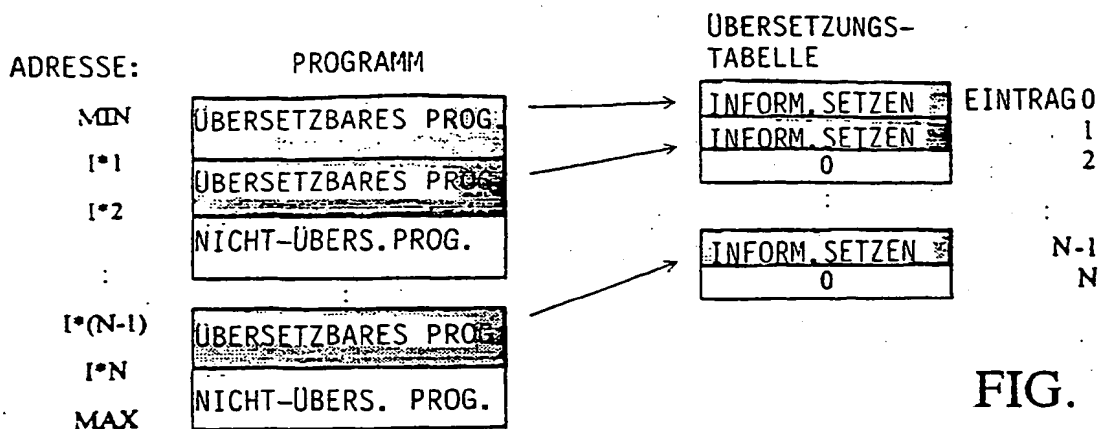


FIG. 26

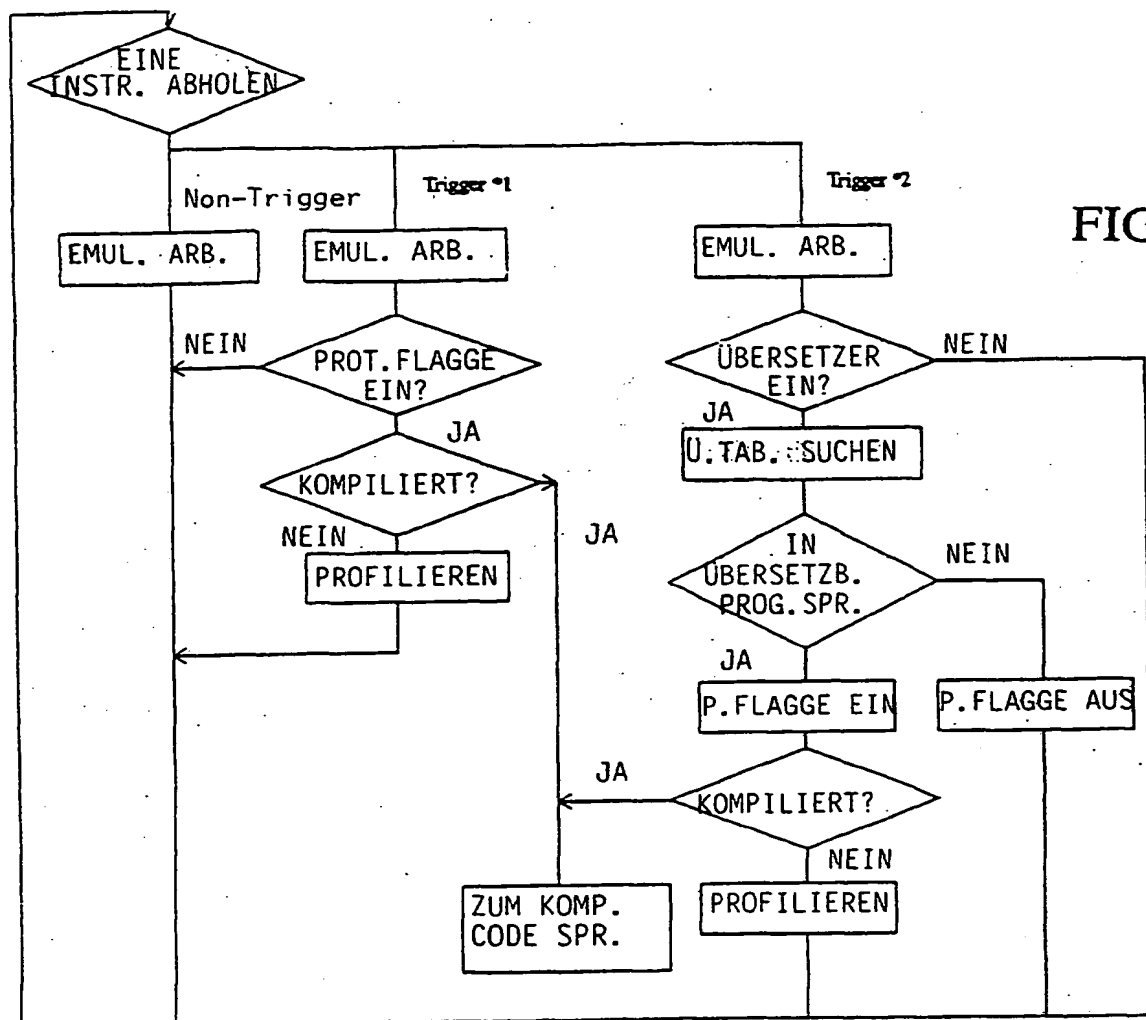


FIG. 27

FIG. 28

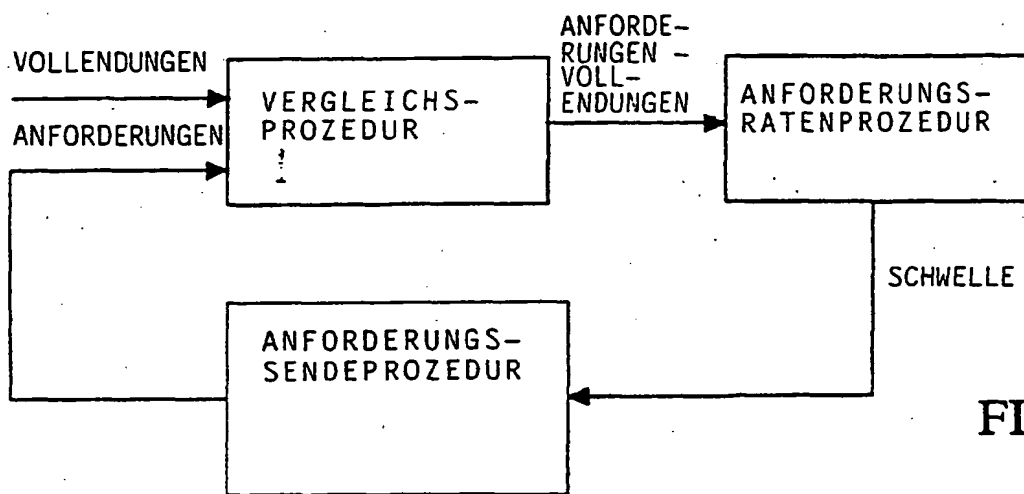
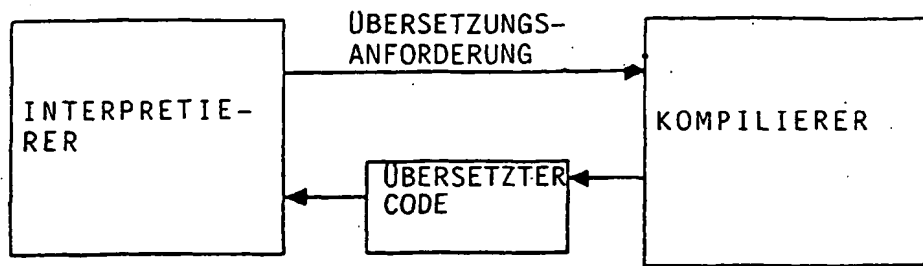
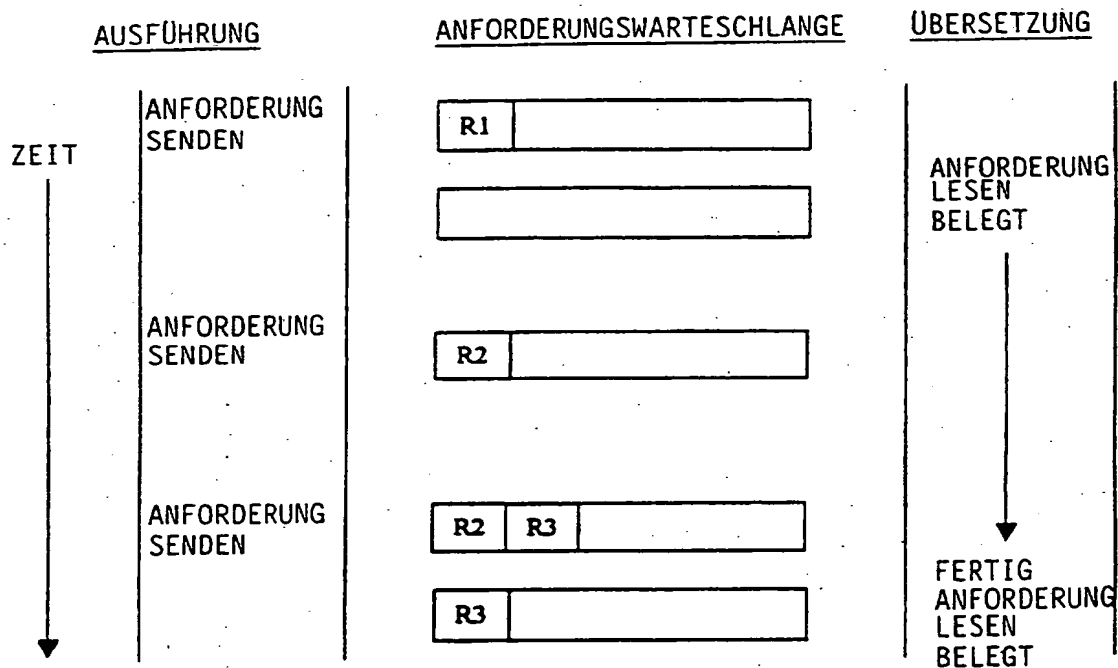


FIG. 29

FIG. 30



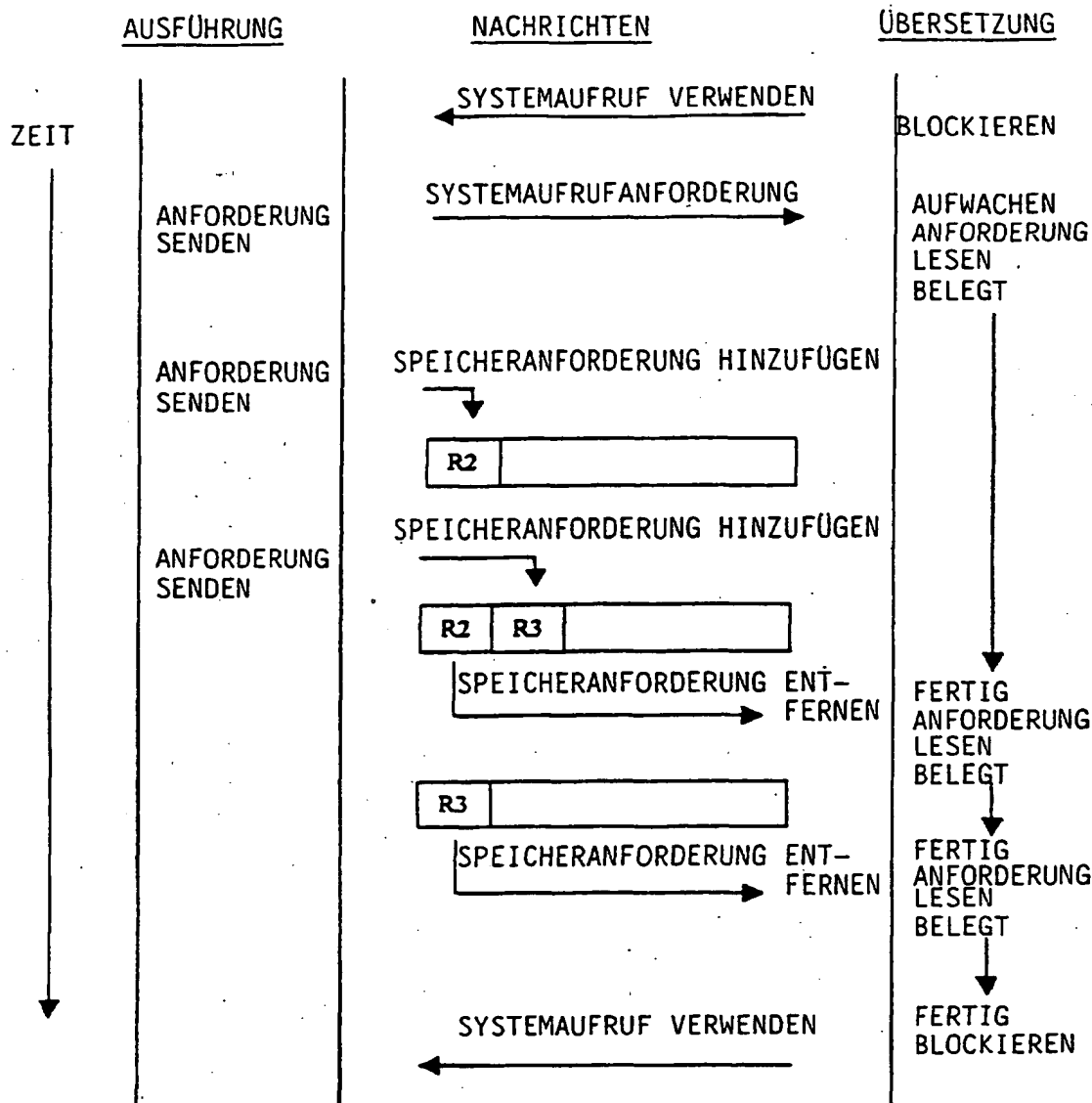


FIG. 31

SEITE 1
(IN PHYSISCHEM
SPEICHER)

SEITE 2
(NICHT IN PHYSISCHEM
SPEICHER)

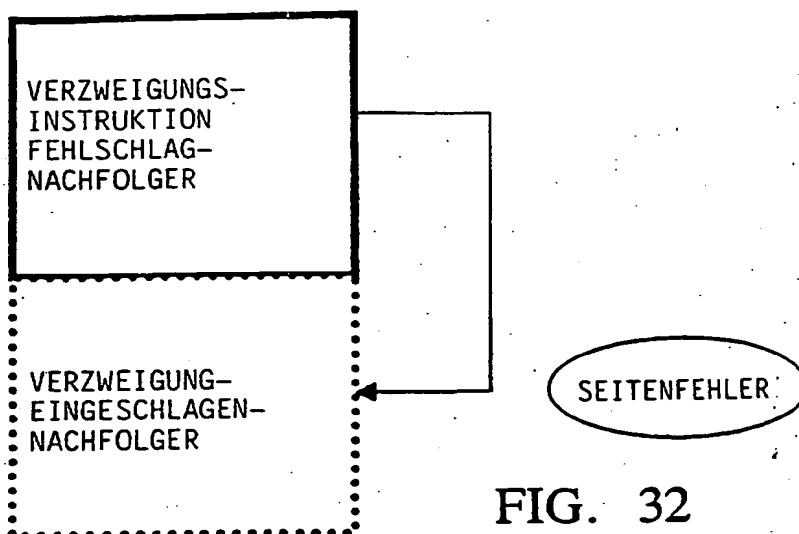


FIG. 32

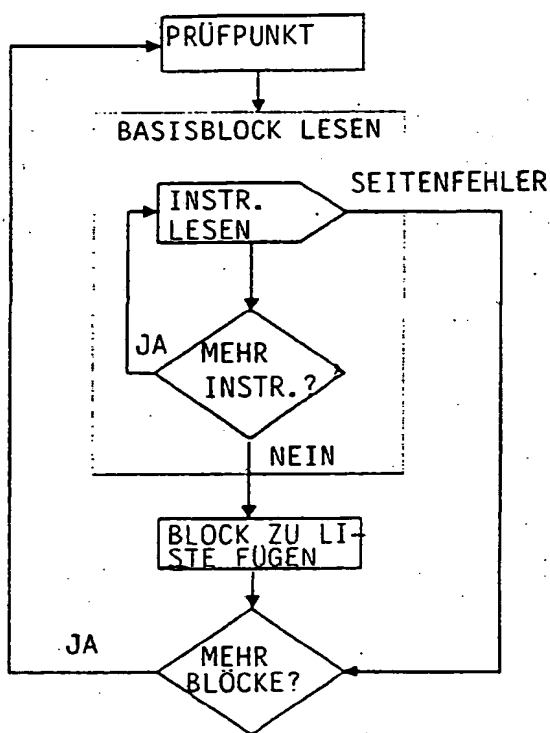


FIG. 33

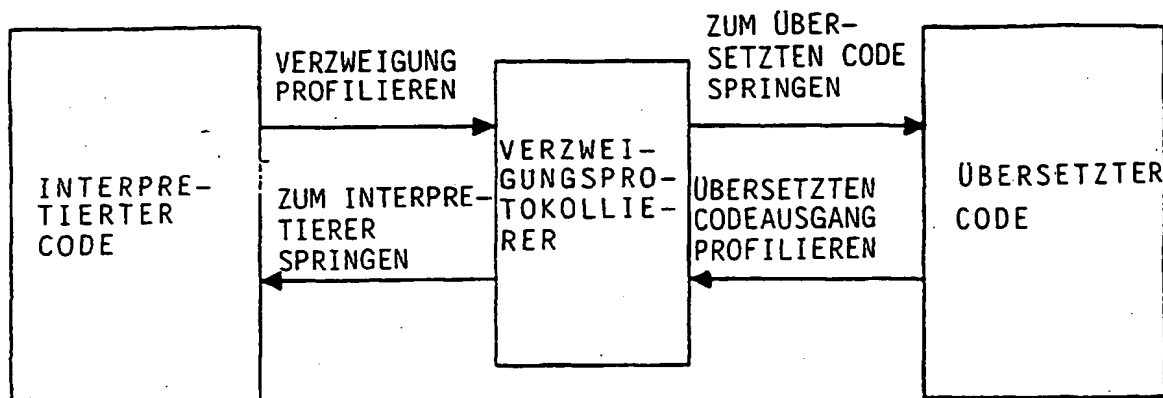


FIG. 34

SATZ ÜBERSETZTER BLÖCKE, WENN SCHWELLE GLEICH 20 %

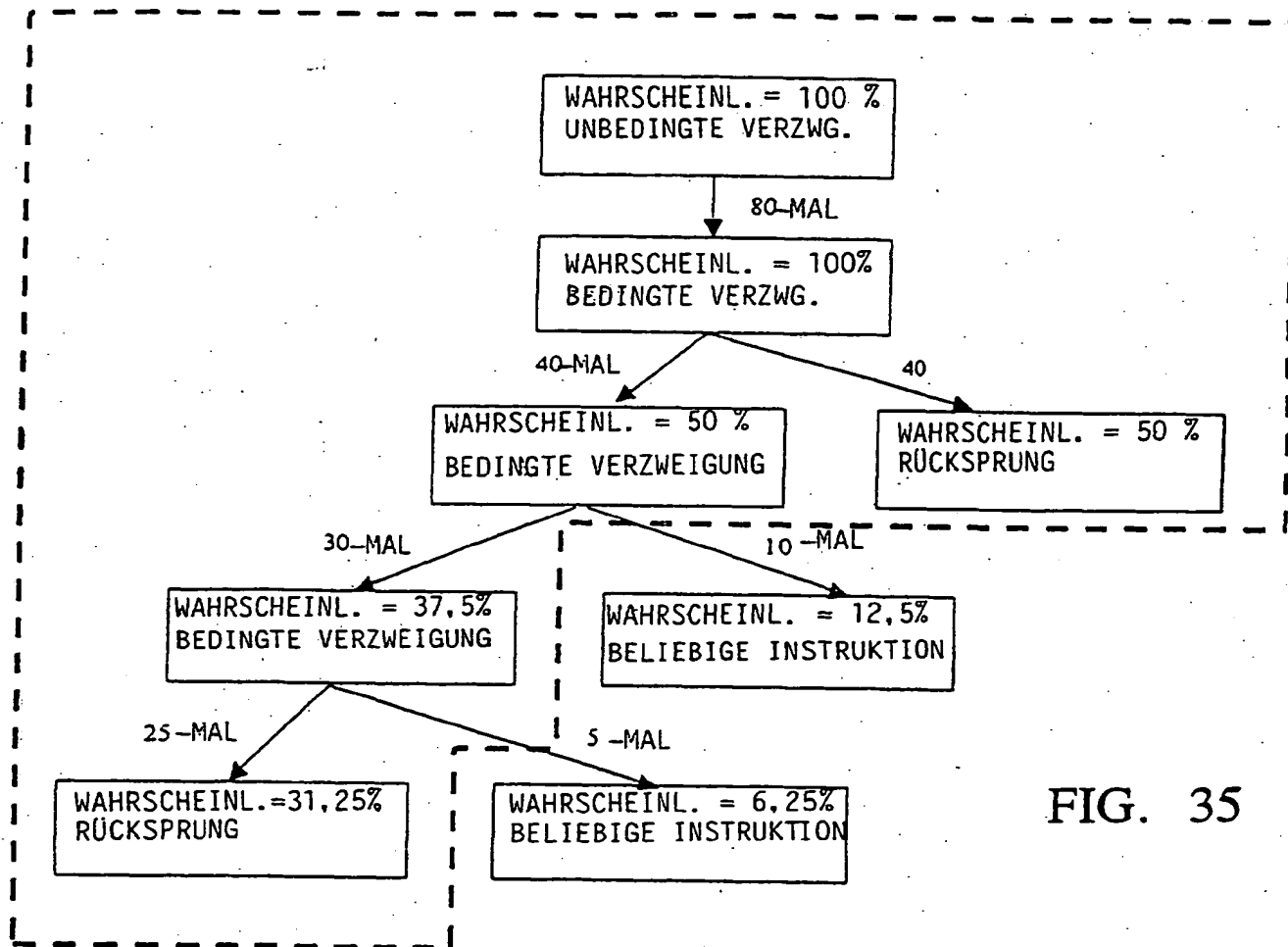


FIG. 35

**This Page is Inserted by IFW Indexing and Scanning
Operations and is not part of the Official Record**

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☒ **BLACK BORDERS**
- ☒ **IMAGE CUT OFF AT TOP, BOTTOM OR SIDES**
- ☒ **FADED TEXT OR DRAWING**
- ☐ **BLURRED OR ILLEGIBLE TEXT OR DRAWING**
- ☒ **SKEWED/SLANTED IMAGES**
- ☒ **COLOR OR BLACK AND WHITE PHOTOGRAPHS**
- ☒ **GRAY SCALE DOCUMENTS**
- ☐ **LINES OR MARKS ON ORIGINAL DOCUMENT**
- ☐ **REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY**
- ☐ **OTHER:** _____

IMAGES ARE BEST AVAILABLE COPY.

As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.

This Page Blank (uspto)